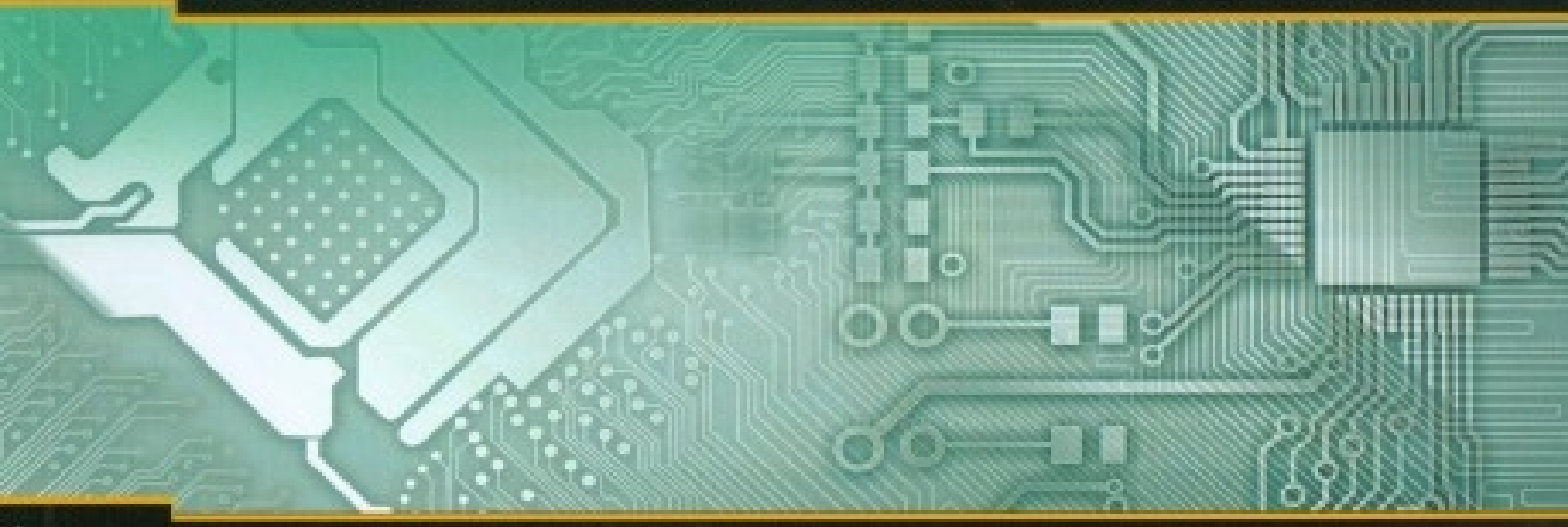
KOSTAS DIMITRIOU Phd & MARKOS HATZITASKOS MSc



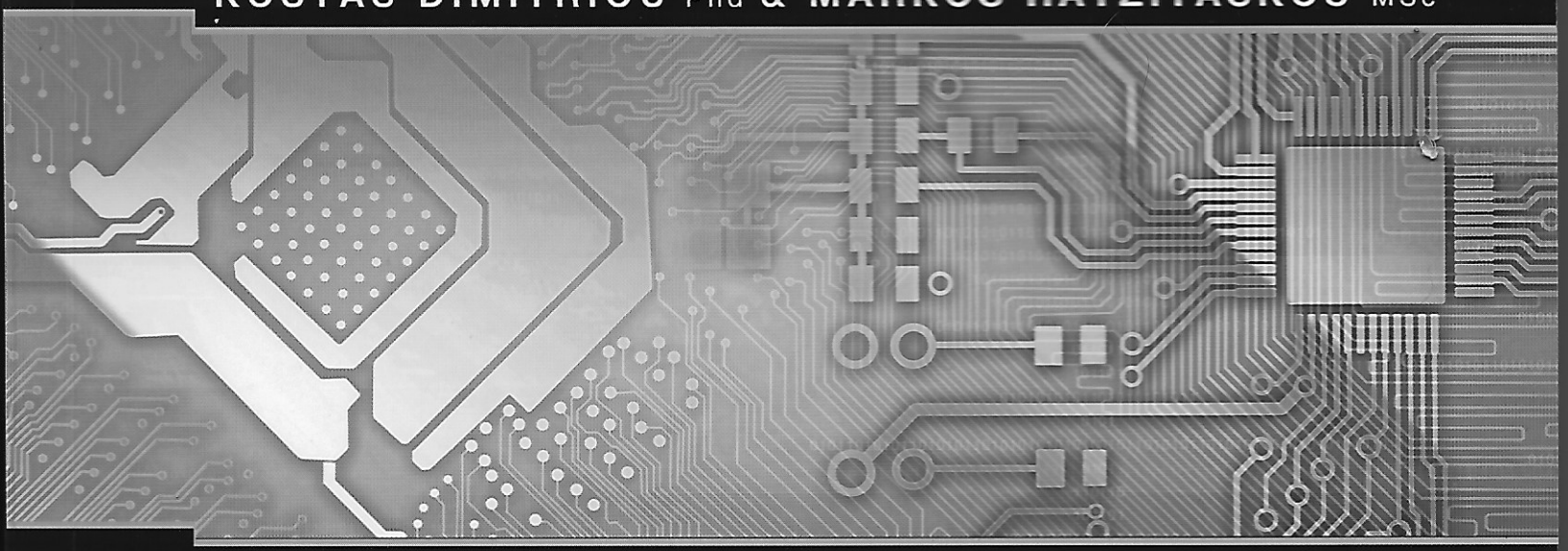
# ADVANCED COMPUTER SCIENCE

For the IB Diploma Program
(International Baccalaureate)

HIGH LEVEL COMPUTER SCIENCE

Express Publishing

KOSTAS DIMITRIOU Phd & MARKOS HATZITASKOS MSc

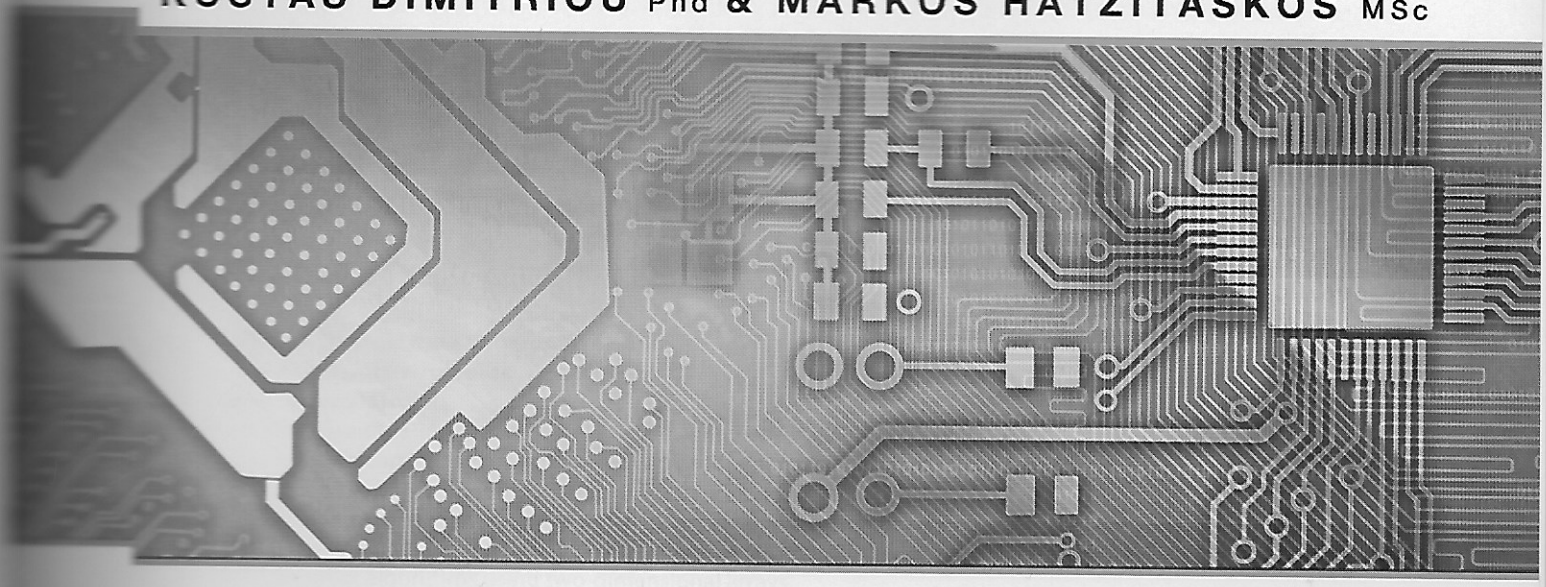


# ADVANCED COMPUTER SCIENCE

For the IB Diploma Program
(International Baccalaureate)

HIGH LEVEL COMPUTER SCIENCE

KOSTAS DIMITRIOU Phd & MARKOS HATZITASKOS MSc



# ADVANCED COMPUTER SCIENCE

For the IB Diploma Program
(International Baccalaureate)

HIGH LEVEL COMPUTER SCIENCE

Express Publishing

**Published by Express Publishing**

**Liberty House, Greenham Business Park, Newbury,**
**Berkshire RG19 6HW, United Kingdom**
**Tel.: (0044) 1635 817 363**
**Fax: (0044) 1635 817 463**
**email: inquiries@expresspublishing.co.uk**
**www.expresspublishing.co.uk**

© Express Publishing, 2016

Design and Illustration © Express Publishing, 2016

First published 2016

Made in EU

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means, electronic, photocopying, or otherwise, without the prior written permission of the publishers.

This book is not meant to be changed in any way.

ISBN 978-1-4715-5233-5

# Copyright page

## List of licensed IB material used:

## DP Computer Science Guide (first exams 2014):

**Topic 5—Abstract data structures**
**5.1 Abstract data structures**
**Thinking recursively**
5.1.1 Identify a situation that requires the use of recursive thinking.
5.1.2 Identify recursive thinking in a specified problem solution.
5.1.3 Trace a recursive algorithm to express a solution to a problem.
**Abstract data structures**
5.1.4 Describe the characteristics of a two-dimensional array.
5.1.5 Construct algorithms using two-dimensional arrays.
5.1.6 Describe the characteristics and applications of a stack.
5.1.7 Construct algorithms using the access methods of a stack.
5.1.8 Describe the characteristics and applications of a queue.
5.1.9 Construct algorithms using the access methods of a queue.
5.1.10 Explain the use of arrays as static stacks and queues.
**Linked lists**
Linked lists will be examined at the level of diagrams and descriptions. Students are not expected to construct linked list algorithms using pseudocode.
5.1.11 Describe the features and characteristics of a dynamic data structure.
5.1.12 Describe how linked lists operate logically.
5.1.13 Sketch linked lists (single, double and circular).
**Trees**
Binary trees will be examined at the level of diagrams and descriptions. Students are not expected to construct tree algorithms using pseudocode. Tracing and constructing algorithms are not expected.
5.1.14 Describe how trees operate logically (both binary and non-binary).
5.1.15 Define the terms: parent, left-child, right-child, subtree, root and leaf.
5.1.16 State the result of inorder, postorder and preorder tree traversal.
5.1.17 Sketch binary trees.
**Applications**
5.1.18 Define the term dynamic data structure.
5.1.19 Compare the use of static and dynamic data structures.
5.1.20 Suggest a suitable structure for a given situation.

**Topic 6—Resource management**
**6.1 Resource management**
**System resources**
6.1.1 Identify the resources that need to be managed within a computer system.
6.1.2 Evaluate the resources available in a variety of computer systems.
6.1.3 Identify the limitations of a range of resources in a specified computer system.
6.1.4 Describe the possible problems resulting from the limitations in the resources in a computer system.
**Role of the operating system**
6.1.5 Explain the role of the operating system in terms of managing memory, peripherals and hardware interfaces.
6.1.7 Outline OS resource management techniques: scheduling, policies, multitasking, virtual memory, paging, interrupt, polling.
6.1.8 Discuss the advantages of producing a dedicated operating system for a device.
6.1.9 Outline how an operating system hides the complexity of the hardware from users and applications.

**Topic 7—Control**
**7.1 Control**
**Centralized control systems**
7.1.1 Discuss a range of control systems.
7.1.2 Outline the uses of microprocessors and sensor input in control systems.
7.1.3 Evaluate different input devices for the collection of data in specified situations.
7.1.4 Explain the relationship between a sensor, the processor and an output transducer.

7.1.5 Describe the role of feedback in a control system.
7.1.6 Discuss the social impacts and ethical considerations associated with the use of embedded systems.

**Distributed systems**

7.1.7 Compare a centrally controlled system with a distributed system.
7.1.8 Outline the role of autonomous agents acting within a larger system.

**D—Object-oriented programming**

**D.4 Advanced program development**

D.4.1 Define the term recursion.
D.4.2 Describe the application of recursive algorithms.
D.4.3 Construct algorithms that use recursion.
D.4.4 Trace recursive algorithms.
D.4.5 Define the term object reference.
D.4.6 Construct algorithms that use reference mechanisms.
D.4.7 Identify the features of the abstract data type (ADT) list.
D.4.8 Describe applications of lists.
D.4.9 Construct algorithms using a static implementation of a list.
D.4.10 Construct list algorithms using object references.
D.4.11 Construct algorithms using the standard library collections included in JETS.
D.4.12 Trace algorithms using the implementations described in assessment statements D.4.9–D.4.11.
D.4.13 Explain the advantages of using library collections.
D.4.14 Outline the features of ADT's stack, queue and binary tree.
D.4.15 Explain the importance of style and naming conventions in code.

KOSTAS DIMITRIOU Phd & MARKOS HATZITASKOS MSc

# ADVANCED COMPUTER SCIENCE

For the IB Diploma Program

(International Baccalaureate)

HIGH LEVEL COMPUTER SCIENCE



Kostas Dimitrou Dedication

*To my son Dimitris and my daughter Eliana.*

*Never forget that I love you.*

Markos Hatzitaskos Dedication

*To all my friends that helped make this book a reality.*

*You know who you are.*

# Preface

Kostas Dimitriou holds a PhD in Spatial Decision Support Systems and Environmental Planning, and has taught computer science courses in various undergraduate and postgraduate University courses. He has participated in many scientific conferences and workshops, twenty research projects, and presented sixty scientific articles. He has been teaching the IB computer science in the Hellenic American Educational Foundation since 2002. He is a Microsoft Certified Educator, Microsoft Expert Educator, Microsoft Expert Education Trainer and Microsoft Innovative Educator Fellow. {kdimitriou@haef.gr}

Markos Hatzitaskos holds an MSc in Advanced Computing and has taught computer science courses throughout all school levels (from primary to high school and the I.B.). He has been teaching in the Hellenic American Educational Foundation since 2011. In his free time, whenever that might be, he develops mobile applications and attends the Athens School of Fine Arts as an undergraduate. {markosh@haef.gr}

The authors would like to thank the Board of Directors and the Administration of the Hellenic American Educational Foundation (HAEF) for providing an ideal working environment. Thanks are also due to Kostas Ziogas who gave us some valuable advice. Both authors would like to express their gratitude to the employees of Express Publishing and especially to our friend Tzeni Vlachou.

The authors would like to acknowledge the ongoing, valuable support of Sophia Arditzoglou, HAEF IB coordinator, over the years. A lot of Computer Science students contributed with valuable ideas, comments and suggestions on early drafts. The computer science class of 2014 encouraged us to start this book.

The purpose of this document is to facilitate learning and help our colleagues and CS students around the world. This book is based on the IB computer science syllabus and follows the IB computer science syllabus. The authors did their very best to cite all resources used. If you find a source that is not properly cited please report it to the authors. This book was inspired by the book[1]: Jones, R & A. Meyenn. (2004). Computer science Java Enabled. International Baccalaureate. Series, IBID press, Victoria.

[1] Jones, R & Meyenn, A. (2004). Computer science Java enabled. International Baccalaureate. Series, IBID press, Victoria.

The following IBO documents were used during the development of this book:


1. International Baccalaureate Organization. (2004). IBDP Computer Science Guide.
2. International Baccalaureate Organization. (2012). IBDP Computer Science Guide.
3. International Baccalaureate Organization. (2012). IBDP Approved notations for developing pseudocode.
4. International Baccalaureate Organization. (2012). IBDP Java Examination Tool Subset.
5. International Baccalaureate Organization. (2012). IBDP Pseudocode in examinations.

# Table of Contents


Topic 5 — Abstract data structures ..... 1

5.1 Abstract data structures ..... 1

Thinking recursively ..... 1

5.1.1 – 5.1.3 Recursive thinking ..... 1

Abstract data structures ..... 7

5.1.4 – 5.1.5 Two dimensional arrays ..... 7

5.1.6 – 5.1.7 Stacks ..... 10

5.1.8 – 5.1.9 Queues ..... 16

5.1.10 Arrays as static stacks and queues ..... 19

Linked lists ..... 30

5.1.11 Features and characteristics of a dynamic data structure ..... 30

5.1.12 Operation of linked lists ..... 31

5.1.13 Sketch linked lists ..... 32

Trees ..... 37

5.1.14 Logical operation of trees ..... 38

5.1.15 Binary-tree related terminology ..... 39

5.1.16 Tree traversal ..... 41

5.1.17 Sketch binary trees ..... 46

Applications ..... 53

5.1.18 Definition of the term dynamic data structure ..... 53

5.1.19 Comparison of static and dynamic data structures ..... 53

5.1.20 Suitable structures ..... 54

End of chapter example questions with answers ..... 56

Chapter References ..... 67

Topic 6 — Resource management ..... 68

6.1 Resource management ..... 68

System resources ..... 68

6.1.1 Identification of critical resources ..... 68

6.1.2 Availability of resources ..... 73

6.1.3 Limitation of resources ..... 76

6.1.4 Problems with insufficient resources ..... 77

Role of the operating system ..... 79

6.1.5 Role of the Operating System (OS) ..... 79

6.1.6 – 6.1.7 OS resource management techniques ..... 82

6.1.8 Dedicated OS for a device ..... 85

6.1.9 OS and complexity hiding ..... 87

End of chapter example questions with answers ..... 89

Chapter References ..... 94

Topic 7 — Control ..... 95

7.1 Control ..... 95

Centralized control systems ..... 95

7.1.1 A range of control systems ..... 95

7.1.2 The uses of microprocessors and sensor input in control systems ..... 101
7.1.3 Different input devices for the collection of data in specified situations ..... 103
7.1.4 The relationship between a sensor, the processor and an output transducer ..... 104
7.1.5 The role of feedback in a control system ..... 106
7.1.6 Social impacts and ethical considerations associated with the use of embedded systems ..... 106
Distributed systems ..... 109
7.1.7 Comparison of centrally controlled systems with distributed systems ..... 109
7.1.8 The role of autonomous agents acting within a larger system ..... 111
End of chapter example questions with answers ..... 115
**Chapter References ..... 120**

**Topic D – Object-oriented programming ..... 121**
**Tool used ..... 121**
**D.4 Advanced program development ..... 122**
D.4.1 The term "recursion" ..... 122
D.4.2 Application of recursive algorithms ..... 122
D.4.3 Construction of algorithms that use recursion ..... 128
D.4.4 Trace of recursive algorithms ..... 129
D.4.5 Define the term object reference ..... 130
D.4.6 Construct algorithms that use reference mechanisms ..... 133
D.4.7 Identify the features of the Abstract Data Type (ADT) list ..... 137
D.4.8 Describe the applications of lists ..... 140
D.4.9 Construct algorithms using a static implementation of a list ..... 144
D.4.10 Construct list algorithms using object references ..... 152
D.4.11 Construct algorithms using the standard library collections included in JETS ..... 161
D.4.12 Trace algorithms using the implementations described in assessment statements D.4.9-D.4.11 ..... 167
D.4.13 Explain the advantages of using library collections ..... 171
D.4.14 Outline the features of ADT's stack, queue and binary tree ..... 173
D.4.15 Explain the importance of style and naming conventions in code ..... 173
End of chapter example questions with answers ..... 177
**Chapter References ..... 284**

**Appendix A — Stacks & Queues ..... 285**
A.1 Stack implementation using the `ArrayList` class ..... 285
A.2 Queue implementation using the `ArrayList` class ..... 287

# TOPIC 5 – ABSTRACT DATA STRUCTURES

Most IB compatible pseudocode examples of this book have been tested using the EZ Pcode practice tool found at:

**https://dl.dropboxusercontent.com/u/275979/ibcomp/pseduocode/pcode.html**

This excellent tool was developed by Mr. Dave Mulkey. The authors wish to express their gratitude to the developer of this valuable educational resource.

© IBO 2012

## Topic 5 — Abstract data structures[1]

### 5.1 Abstract data structures

#### Thinking recursively

##### 5.1.1 – 5.1.3 Recursive thinking

| Exit skills. Students should be able to[1]: |
| --- |
| Identify a situation that requires the use of recursive thinking. Identify recursive thinking in a specified problem solution. Trace a recursive algorithm to express a solution to a problem. |

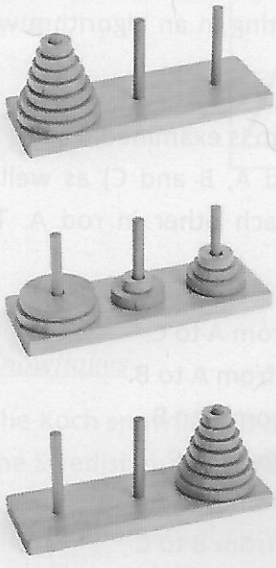


Image 5.1: The Towers of Hanoi game

Recursion is when a method calls itself until some terminating condition is met. This is accomplished **without** any specific repetition construct, such as a **while** or a **for** loop. Recursion follows one of the basic problem solving techniques, which is to break down the problem at hand into smaller subtasks. Any algorithm that may be presented in a recursive manner can also be presented in an iterative manner and vice versa. In most cases, recursive algorithms are considered as harder to code.

*Towers of Hanoi*[2]

In order to gain a firm understanding of the basic idea, as well as the application of recursion, the following example

[1] International Baccalaureate Organization. (2012). IBDP Computer Science Guide.

[2] Towers of Hanoi. (2015, November 17). In *Wikipedia, The Free Encyclopedia*. Retrieved 14:03, November 17, 2014, from https://en.wikipedia.org/wiki/Tower_of_Hanoi

presents what is known as the **Towers of Hanoi**. The Towers of Hanoi is a puzzle that consists of three rods and a number of discs of different sizes, which can slide onto any rod. The puzzle starts with the discs in a neat stack in ascending order of size on the first rod, the smallest at the top, as shown in Image 5.1. The goal of the puzzle is to move the stack of discs from the first rod to the third rod, obeying the following rules:

- A disc may not be placed on top of a smaller one.
- Only one disc may move on every move.
- A disc may not be moved if it is not the top disc on a stack.
- For temporary storage, the third rod may be used.

There are various approaches that can solve the Towers of Hanoi problem, including both iterative and recursive solutions. We will be concentrating on a recursive solution, by recognizing that this puzzle may be solved by breaking it into smaller and smaller similar puzzles, until a solution is reached.

Assume that the rods are named A, B and C and that *n* represents the number of discs (with 1 being the smallest, at the top, and n being the largest, at the bottom). A recursive solution to the Tower of Hanoi problem, in order to move *n* discs from rod A to rod C could be the following:

- Move *n-1* discs from rod A to rod B, leaving disc *n* in rod A.
- Move disc *n* from rod A to rod C.
- Move *n-1* discs from rod B to rod C.

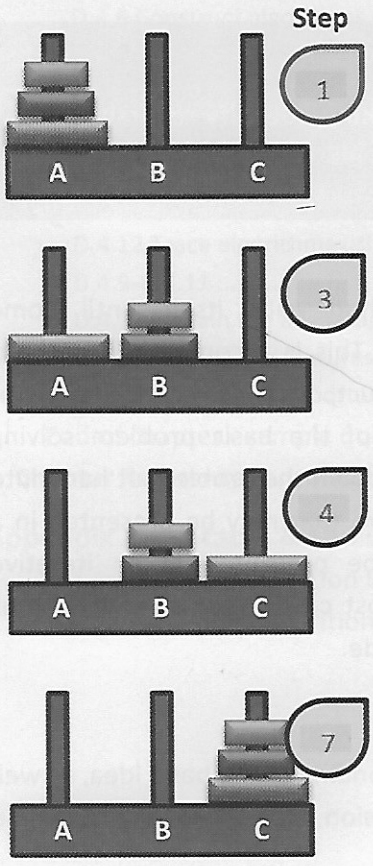



**Figure 5.1: Steps of the game**

The algorithm above is recursive as it is applied again and again in both the first and the third steps for *n-1* discs. At some point *n* will be equal to 1 and a single disc will be moved from rod A to rod C, resulting in an algorithm with finite number of steps.

A working example of this algorithm is examined. Figure 5.1 represents the three rods (named A, B and C) as well as three discs, stacked on top of each other in rod A. The algorithm goes as follows:

1. Move green disc from A to C.
2. Move orange disk from A to B.
3. Move green disk from C to B
4. Move grey disk from A to C
5. Move green disk from B to A
6. Move orange disk from B to C
7. Move green disc from A to C

The recursive algorithm for the solution of the Towers of Hanoi problem is also presented in Figure 5.2. Pay attention to the fact that a sub-procedure called moveDiscs is used. moveDiscs takes four

arguments. The number of the discs (n), the rod the discs are to be moved from (from), the rod to which the discs are to be moved to (dest), as well as the rod that will not be used (aux). The arguments of the moveDiscs sub-procedure (that is, n, from, aux, dest) should not be confused with the name of the rods used previously (A, B and C).

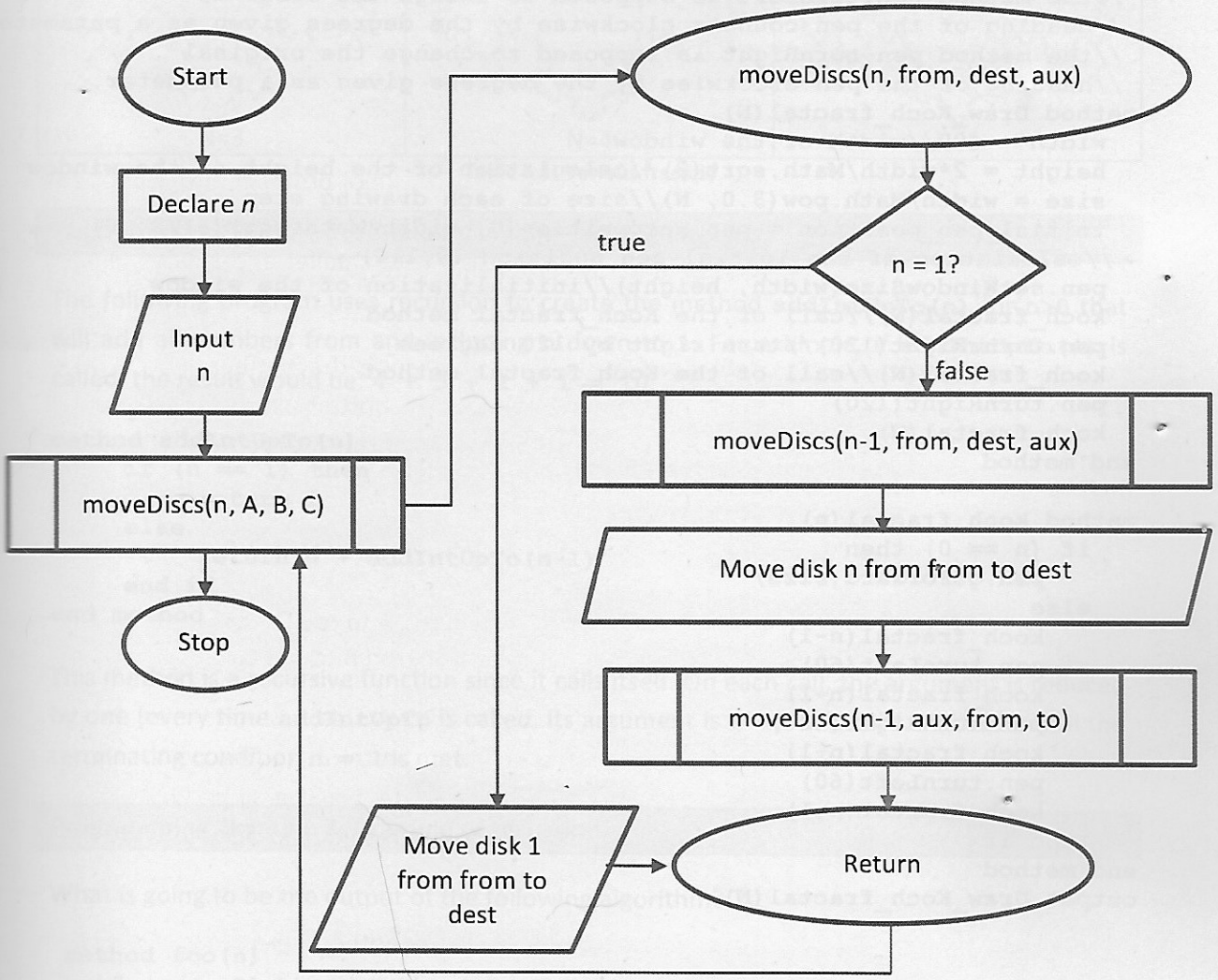



**Figure 5.2: The Towers of Hanoi flowchart**

*Snowflakes*

The Koch snowflake is a mathematical curve which is based on the Koch curve, developed by the Swedish mathematician Helge von Koch.

This mathematical curve can be constructed by starting with an equilateral triangle. Using recursion each line segment changes using the following steps:

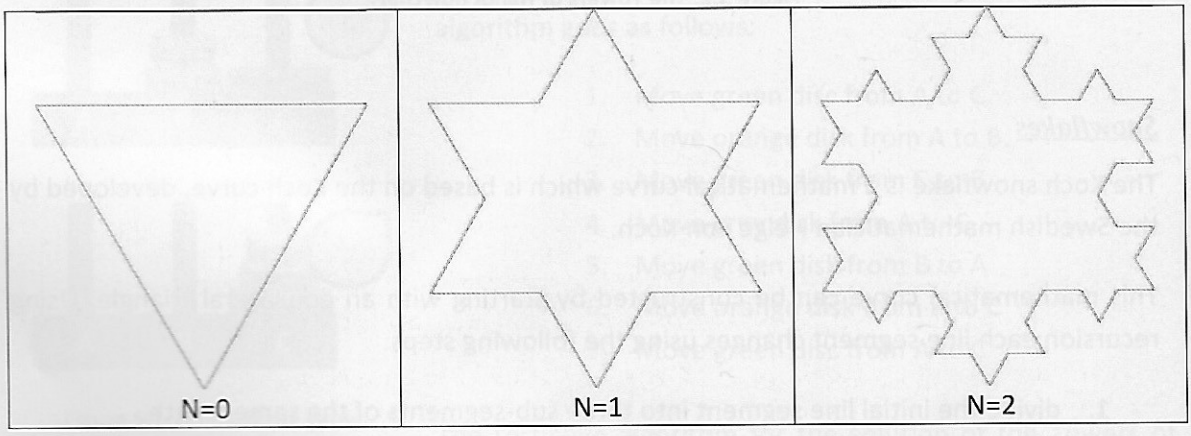
1. divide the initial line segment into three sub-segments of the same length.
2. draw an outward pointing equilateral triangle that has the middle segment from step (1) as its base.
3. delete the line segment that is the base of the triangle from previous step.

The following algorithm expressed in IB pseudocode creates a 400 by 461 window and draws a Koch fractal:

```
//Three curves that shape an equilateral triangle
//pen originally is heading at 90 degrees (x axis)
//the method pen.goForward is supposed to control
//a pen that plots line segments on the screen
//the method pen.turnLeft is supposed to change the original
//heading of the pen counter clockwise by the degrees given as a parameter.
//the method pen.turnRight is supposed to change the original
//heading of the pen clockwise by the degrees given as a parameter.
method Draw_Koch_fractal(N)
  width = 400//width of the window
  height = 2*width/Math.sqrt(3)//calculation of the height of the window
  size = width/Math.pow(3.0, N)//size of each drawing step
  initial_pen_position = pen.setposition(0, width*Math.sqrt(3)/2, 0)
  //calculation of the initial pen position (0,114)
  pen.setWindowSize(width, height)//initialization of the window
  koch_fractal(N)//call of the Koch_fractal method
  pen.turnrRight(120)//turn right by 120 degrees
  koch_fractal(N)//call of the Koch_fractal method
  pen.turnRight(120)
  koch_fractal(N)
end method

method koch_fractal(n)
   if (n == 0) then
      pen.goForward(size)
   else
      koch_fractal(n-1)
      pen.turnLeft(60)
      koch_fractal(n-1)
      pen.turnRight(120)
      koch_fractal(n-1)
      pen.turnLeft(60)
      koch_fractal(n-1)
    end if
end method
output Draw_Koch_fractal(N)
```

The following table depicts the snowflakes produced by the above algorithm for N=0 to 5:



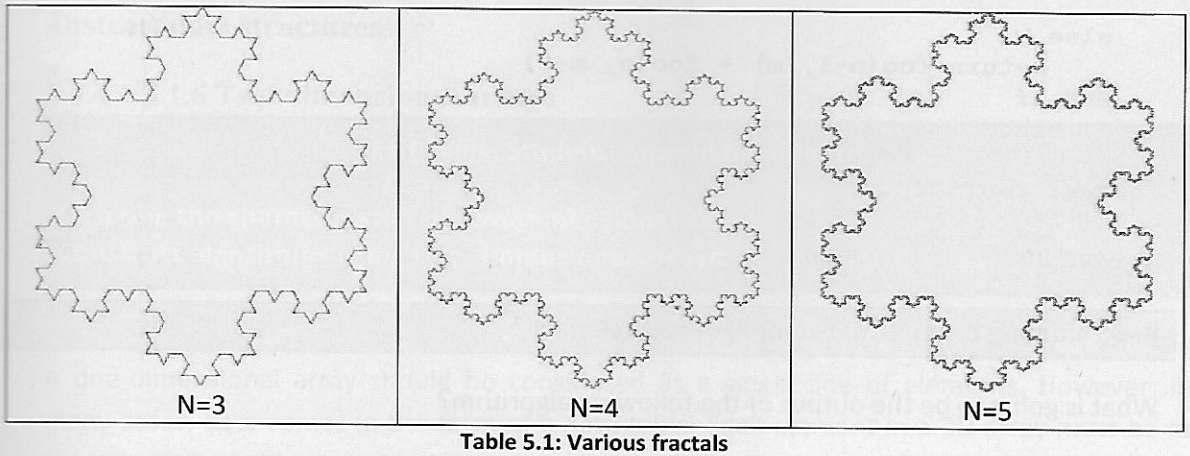



**Table 5.1: Various fractals**

## Programming Example 1: Simple recursion – adding integers.

The following program uses recursion to create the method `addIntUpTo(n)` for `n>0` that will add all numbers from and including `n` down to 1. For example, if `addIntUpTo(4)` is called, the result would be: `4 + 3 + 2 + 1 = 10`

```
method addIntUpTo(n)
     if (n == 1) then
          return 1
     else
          return n + addIntUpTo(n-1)
     end if
end method
```

This method is a recursive function since it calls itself. On each call, the argument is reduced by one (every time `addIntUpTo` is called, its argument is `n-1`). `n-1` calls are made until the terminating condition `n = 1` is met.

## Programming Example 2: Example of recursion.

What is going to be the output of the following algorithm?

```
method foo(n)
  if (n <= 1) then
      return 1
  else
      return foo(n-1) + foo(n-2)
  end if
end method

output foo(5)
```

Answer: 8

## Programming Example 3: Example of recursion.

What is going to be the output of the following algorithm?

```
method foo(n, m)
  if (n <= 1) OR (m <= 1) then
      return 2
```

```
  else
      return foo(n-1, m) + foo(n, m-2)
  end if
end method

output foo(5,4)
```

Answer: 30

## Programming Example 4: Example of recursion.

What is going to be the output of the following algorithm?

```
method foo(n, m)
  output "value of n=", n, "value of m =", m
  if (n <= 1) OR (m<=1) then
      return 2
  else
      return foo(n-1, m-n)+foo(n, m-2)
  end if
end method

output "Output is", foo(3,2)
```

Answer:

```
value of n= 3 value of m = 2
value of n= 2 value of m = -1
value of n= 3 value of m = 0
Output is 4
```

## Programming Example 5: Example of recursion.

What is going to be the output of the following algorithm?

```
method Foo(X,Y)
 if X < Y then
  return Foo(X+1,Y-2)
 else if X = Y then
  return 2*Foo(X+2,Y-3)-3
 else
  return 2*X+3*Y
 end if
end method
output "Output is", Foo(3,12)
```

Answer:

```
Output is 47
```

## 5.1.4 – 5.1.5 Two dimensional arrays

| Exit skills. Students should be able to[1]: |
|---|
| Describe the characteristics of two-dimensional arrays.<br>Construct algorithms using two-dimensional arrays. |

A one-dimensional array should be considered as a single line of elements. However, in many cases, data comes in the form of a data table. Each element in a 2D array must be of the same type, either a primitive or object type. Take, for example, five exam scores of a student, as a data record, and represent it as a row of information. The data records for ten students, would then be a table of 10 rows. Below is the visualization of this collection of data:

*A lot of information and examples of two-dimensional arrays can be found in the book Core Computer Science for the IB Diploma Program[3].*

| | | Exam 1 | Exam 2 | Exam 3 | ... | Exam 5 |
|---|---|---|---|---|---|---|
| | | *Index 0* | *Index 1* | *Index 2* | | *Index 4* |
| **Student 1** | *Index 0* | 98 | 68 | 65 | | 67 |
| **Student 2** | *Index 1* | 77 | 77 | 88 | | 90 |
| ... | ... | ... | ... | ... | | ... |
| **Student 10** | *Index 9* | 88 | 86 | 90 | | 81 |

**Table 5.2: Two dimensional array Scores**

2D arrays are indexed by two subscripts. The indices must be integers. The first one refers to the row, while the second to the column. `Scores[1][1]` refers to Exam 2 of the second student. Its value is 77.

**Programming Example 6: Two dimensional array (exam scores).**

```
//This program will use the array Scores which is a 2D ARRAY.
//It will print the contents of the array.
//5 students with 5 exams each
Scores =
[[98,68,65,73,67],
[77,77,88,78,90],
[53,63,74,85,72],
[77,77,68,78,91],
[88,86,90,56,81]]
STUDENT = 0
EXAM = 0
loop STUDENT from 0 to 4
       output STUDENT +1, "Student"
    loop EXAM from 0 to 4
       output "----", "Exam ", EXAM+1, Scores[STUDENT][EXAM]
    end loop
end loop
```

OUTPUT

```
1 Student
---- Exam 1 98
---- Exam 2 68
---- Exam 3 65
---- Exam 4 73
---- Exam 5 67
2 Student
---- Exam 1 77
---- Exam 2 77
---- Exam 3 88
---- Exam 4 78
---- Exam 5 90
3 Student
---- Exam 1 53
---- Exam 2 63
---- Exam 3 74
---- Exam 4 85
---- Exam 5 72
4 Student
---- Exam 1 77
---- Exam 2 77
---- Exam 3 68
---- Exam 4 78
---- Exam 5 91
5 Student
---- Exam 1 88
---- Exam 2 86
---- Exam 3 90
---- Exam 4 56
---- Exam 5 81
```

**Programming Example 7: Finds if a grade has the number 5 as its last digit.**

```
Scores =
[[98,68,65,73,67],
[77,77,88,78,90],
[53,65,74,85,72],
[77,77,68,78,91],
[88,86,90,56,81]]
STUDENT = 0
EXAM = 0
loop STUDENT from 0 to 4
       output STUDENT +1, "Student"
    loop EXAM from 0 to 4
       if (Scores[STUDENT][EXAM] mod 10 = 5) then
        output "----", "Exam ", EXAM+1, Scores[STUDENT][EXAM]
       end if
    end loop
end loop
```

OUTPUT

```
1 Student
---- Exam 3 65
2 Student
3 Student
---- Exam 2 65
---- Exam 4 85
4 Student
5 Student
```

**Programming Example 8: Finds and outputs the number of "8"s each score contains. It also outputs the total number of appearance of digit "8".**

```
Scores =
[[98,68,65,73,67],
[77,77,88,78,90],
[77,77,88,78,91],
[88,86,90,56,81]]
STUDENT = 0
EXAM = 0
TCOUNTER = 0
loop STUDENT from 0 to 3
       output STUDENT +1, "Student"
    loop EXAM from 0 to 4
         X = 1
         COUNTER = 0
         X = Scores[STUDENT][EXAM]
      loop while X>0
         if  X mod 10 = 8 then
            COUNTER = COUNTER + 1
            TCOUNTER = TCOUNTER +1
         end if
             X = div (X, 10)
      end while
         output "----", "The grade of exam ", EXAM+1, "has",
COUNTER, "eight(s)"
    end loop
end loop
output " A total of ", TCOUNTER, "eights appear in all grades"
```

OUTPUT:

```
1 Student
---- The grade of exam 1 has 1 eight(s)
---- The grade of exam 2 has 1 eight(s)
---- The grade of exam 3 has 0 eight(s)
---- The grade of exam 4 has 0 eight(s)
---- The grade of exam 5 has 0 eight(s)
2 Student
---- The grade of exam 1 has 0 eight(s)
---- The grade of exam 2 has 0 eight(s)
---- The grade of exam 3 has 2 eight(s)
---- The grade of exam 4 has 1 eight(s)
---- The grade of exam 5 has 0 eight(s)
3 Student
---- The grade of exam 1 has 0 eight(s)
```

```
---- The grade of exam 2 has 0 eight(s)
---- The grade of exam 3 has 2 eight(s)
---- The grade of exam 4 has 1 eight(s)
---- The grade of exam 5 has 0 eight(s)
4 Student
---- The grade of exam 1 has 2 eight(s)
---- The grade of exam 2 has 1 eight(s)
---- The grade of exam 3 has 0 eight(s)
---- The grade of exam 4 has 0 eight(s)
---- The grade of exam 5 has 1 eight(s)
A total of 12 eights appear in all grades
```

### 5.1.6 - 5.1.7 Stacks

| Exit skills. Students should be able to[1]: |
|---|
| Describe the characteristics and applications of a stack.<br>Construct algorithms using the access methods of a stack.<br>Trace algorithms that use stacks. |

*Characteristics*

A stack stores a set of elements in a particular order and allows access only to the last item inserted. Items are retrieved in the reverse order in which they are inserted. The stack is a Last-In, First-Out data (LIFO) structure. The elements of a stack may be numbers, Boolean values, characters, objects, arrays, strings, etc.

Stacks utilize three methods:

1. **`push()`**. Pushes an item onto a stack.
2. **`pop()`**. Removes and returns the last item entered in the stack.
3. **`isEmpty()`**. Tests if a stack is empty. It will return true if stack contains no elements.

Suppose we want to add the elements 5, 4, 3 in a stack named Numbers. The following diagram explains this situation:

| | |
|---|---|
|  | **Stack is initially empty.** |
|  | **`Numbers.push(5)`**<br>5 was added to the stack. |
|  | **`Numbers.push(4)`**<br>4 was added to the stack |

| | |
|---|---|
| 3<br>4<br>5 | `Numbers.push(3)`<br>3 was added to the stack |

Suppose we want to remove all the elements from the stack.

The following example presents this situation:

| | |
|---|---|
| 3<br>4<br>5 | **Stack contains 3 numbers.** |
| <br>4<br>5 | `X = Numbers.pop()`<br>Top element was removed from the stack. This element was the number 3. 3 was assigned to variable `X` |
| <br><br>5 | `X = Numbers.pop()`<br>Top element was removed from the stack. This element was the number 4. 4 was assigned to variable `X` |
| <br><br> | `X = Numbers.pop()`<br>Top element was removed from the stack. This element was the number 5. 5 was assigned to variable `X`. Stack is empty. |

*Applications*

- The back button of a web browser uses a stack to function. Every time a URL is visited it is stored on a stack. The last address that was visited is on the top of the stack. The first address that was visited during the current web session is on the bottom of the stack. If one selects the Back button, he/she begins to visit the previous pages they have visited in reverse order.
- Microprocessors usually use a stack to handle methods. Suppose a method, `A`, which returns an integer, with parameters `b` and `c` of type integer, is called. In Java this would look like this:

```
int c = A(a, b);
```

The method header would look like this:

```
public static int A(int b, int c)
```

The method body should look like this:

```
{method body
  return r}
```

When `A` is called, its return address, as well as `b` and `c` are pushed onto the microprocessors stack. When the method returns `r`, the return address and the parameters (arguments) are popped off the stack. The overall process is more complicated, but further explanation is beyond the scope of this book.

- Recursive methods also utilize the system stack to keep track of each recursive call. This block of memory is used to store temporary data required for program execution. The calls are nested inside each other. Initially, all recursive calls are unfolded and pushed onto the stack, until the base case is reached and then all recursive calls are popped from the stack, when necessary. In the following example the left-hand code fragment will return 4. The right-hand code fragment will generate a run time error because the recursive program will never reach the terminating condition.

```
public class Rec_Demo
{
 public static int question(int n)
 {
  if (n <= 0)
   return -2;
  else
   return (question(n-100)+3);
 }

 public static void main(String[]
a)
 {
   int l = question(111);
   System.out.println("l= "+l);
 }
}
```

OUTPUT
l=4

```
public class Rec_Demo
{
 public static int question(int n)
 {
  if (n <= 0)
   return -2;
  else
   return (question(n+100)+3);
 }

 public static void main(String[]
a)
 {
  int l = question(111);
  System.out.println("l= " + l);
 }
}
```

OUTPUT
java.lang.StackOverflowError:null

*Algorithms*

**Programming Example 9: Use of a stack, an array and a collection.**

```
//==== Reverse and store ==============================
// This algorithm uses an array, a stack and a collection.
// It reads names from the array, reverses them,
// using the stack, and stores the contents of the stack
// inside the collection.
// ====================================================

NAMES = ["Kostas", "Markos", "Anna", "Mary", "Takis"]
NAMES_C = new Collection()
STACK_NAMES = new Stack()
I = 0

loop I from 0 to 4
   STACK_NAMES.push(NAMES[I])
end loop

output "Add names in the collection:"
output "============================"
loop while NOT(STACK_NAMES.isEmpty())
   NAME = STACK_NAMES.pop()
   NAMES_C.addItem(NAME)
   output NAME, "was entered in the collection"
end loop
```

```
output ""
output "Names stored in the collection:"
output "============================"
loop while NAMES_C.hasNext()
   output NAMES_C.getNext()
end loop
```

OUTPUT

```
Add names in the collection:
============================
Takis was entered in the collection
Mary was entered in the collection
Anna was entered in the collection
Markos was entered in the collection
Kostas was entered in the collection

Names stored in the collection:
============================
Takis
Mary
Anna
Markos
Kostas
```

## Programming Example 10: Use of stacks to implement the Towers of Hanoi algorithm

The following program uses 4 stacks to solve the Towers of Hanoi problem:

```
//Declaration and initialization of variables
SPa = new Stack() //a new stack
SPb = new Stack() //a new stack
SPc = new Stack() //a new stack
SPd = new Stack() //a new stack
daa = new Array() //auxiliary array to use in display method
dbb = new Array() //auxiliary array to use in display method
dcc = new Array() //auxiliary array to use in display method
PEGS = [SPa, SPb, SPc, SPd] //an array of four stacks
I = 0
a = 1
b = 2
c = 3
t = 0
n = 0
m = 0
da = ""
db = ""
dc = ""
NUM = 5 //the number of disks
f1 =" "
f2 =" "
f3 =" "

TowersofHanoi(NUM) //the number of disks
```

```
//The following method is the starting point
//of the program
method TowersofHanoi(n)
 loop I from 0 to n-1
    m = n-I
    PEGS[1].push(m)
 end loop
 display()
 move(n,1,2,3)
end method

//This is a recursive method used to solve the problem
method move(n, a, b, c)
if n>0 then
   move(n-1, a, c, b)
   t = PEGS[a].pop()
   PEGS[c].push(t)
   display()
   move(n-1, b, a, c)
 end if
end method

//The following method is used to visualize the
//pegs and the disks. Three auxiliary arrays are
//used so as to display the contents of each stack.
method display()
  output ""
  output " | A | B | C |"
 loop I from 0 to NUM-1
  daa[I] = PEGS[1].pop()//put the elements of the stack to an array
  dbb[I] = PEGS[2].pop()//put the elements of the stack to an array
  dcc[I] = PEGS[3].pop()//put the elements of the stack to an array
 end loop

 loop I from 0 to NUM-1
  da = daa[I]
  db = dbb[I]
  dc = dcc[I]
  f1 = String(da)//covert to string
  if f1 == "null" then
    f1="-"
  end if
  f2 = String(db) //covert to string
  if f2 == "null" then
    f2="-"
  end if
  f3 = String(dc) //covert to string
  if f3 == "null" then
     f3="-"
  end if
  output " | ", f1, " | ", f2, " | ", f3, " | "
 end loop

 loop I from 0 to NUM-1
  m = NUM-1-I
  PEGS[1].push(daa[m]) //put the elements of the array daa back to the stack
  PEGS[2].push(dbb[m]) //put the elements of the array dbb back to the stack
  PEGS[3].push(dcc[m]) //put the elements of the array dcc back to the stack
 end loop
end method
```

FORMATED OUTPUT

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | A | B | C | 2 | A | B | C | 3 | A | B | C | 4 | A | B | C |
| | 1 | – | – | | 2 | – | 1 | | 3 | 2 | 1 | | 3 | 1 | – |
| | 2 | – | – | | 3 | – | – | | 4 | – | – | | 4 | 2 | – |
| | 3 | – | – | | 4 | – | – | | 5 | – | – | | 5 | – | – |
| | 4 | – | – | | 5 | – | – | | – | – | – | | – | – | – |
| | 5 | – | – | | – | – | – | | – | – | – | | – | – | – |
| 5 | A | B | C | 6 | A | B | C | 7 | A | B | C | 8 | A | B | C |
| | 4 | 1 | 3 | | 1 | 2 | 3 | | 1 | – | 2 | | 4 | – | 1 |
| | 5 | 2 | – | | 4 | – | – | | 4 | – | 3 | | 5 | – | 2 |
| | – | – | – | | 5 | – | – | | 5 | – | – | | – | – | 3 |
| | – | – | – | | – | – | – | | – | – | – | | – | – | – |
| | – | – | – | | – | – | – | | – | – | – | | – | – | – |
| 9 | A | B | C | 10 | A | B | C | 11 | A | B | C | 12 | A | B | C |
| | 5 | 4 | 1 | | 5 | 1 | 2 | | 2 | 1 | 3 | | 1 | 4 | 3 |
| | – | – | 2 | | – | 4 | 3 | | 5 | 4 | – | | 2 | – | – |
| | – | – | 3 | | – | – | – | | – | – | – | | 5 | – | – |
| | – | – | – | | – | – | – | | – | – | – | | – | – | – |
| | – | – | – | | – | – | – | | – | – | – | | – | – | – |
| 13 | A | B | C | 14 | A | B | C | 15 | A | B | C | 16 | A | B | C |
| | 1 | 3 | – | | 2 | 3 | 1 | | 5 | 2 | 1 | | 5 | 1 | – |
| | 2 | 4 | – | | 5 | 4 | – | | – | 3 | – | | – | 2 | – |
| | 5 | – | – | | – | – | – | | – | 4 | – | | – | 3 | – |
| | – | – | – | | – | – | – | | – | – | – | | – | 4 | – |
| | – | – | – | | – | – | – | | – | – | – | | – | – | – |
| 17 | A | B | C | 18 | A | B | C | 19 | A | B | C | 20 | A | B | C |
| | – | 1 | 5 | | 1 | 2 | 5 | | 1 | 3 | 2 | | – | 3 | 1 |
| | – | 2 | – | | – | 3 | – | | – | 4 | 5 | | – | 4 | 2 |
| | – | 3 | – | | – | 4 | – | | – | – | – | | – | – | 5 |
| | – | 4 | – | | – | – | – | | – | – | – | | – | – | – |
| | – | – | – | | – | – | – | | – | – | – | | – | – | – |
| 21 | A | B | C | 22 | A | B | C | 23 | A | B | C | 24 | A | B | C |
| | 3 | 4 | 1 | | 3 | 1 | 2 | | 2 | 1 | 5 | | 1 | 4 | 5 |
| | – | – | 2 | | – | 4 | 5 | | 3 | 4 | – | | 2 | – | – |
| | – | – | 5 | | – | – | – | | – | – | – | | 3 | – | – |
| | – | – | – | | – | – | – | | – | – | – | | – | – | – |
| | – | – | – | | – | – | – | | – | – | – | | – | – | – |
| 25 | A | B | C | 26 | A | B | C | 27 | A | B | C | 28 | A | B | C |
| | 1 | – | 4 | | 2 | – | 1 | | 3 | 2 | 1 | | 3 | 1 | 4 |
| | 2 | – | 5 | | 3 | – | 4 | | – | – | 4 | | – | 2 | 5 |
| | 3 | – | – | | – | – | 5 | | – | – | 5 | | – | – | – |
| | – | – | – | | – | – | – | | – | – | – | | – | – | – |
| | – | – | – | | – | – | – | | – | – | – | | – | – | – |
| 29 | A | B | C | 30 | A | B | C | 31 | A | B | C | 32 | A | B | C |
| | – | 1 | 3 | | 1 | 2 | 3 | | 1 | – | 2 | | – | – | 1 |
| | – | 2 | 4 | | – | – | 4 | | – | – | 3 | | – | – | 2 |
| | – | – | 5 | | – | – | 5 | | – | – | 4 | | – | – | 3 |
| | – | – | – | | – | – | – | | – | – | 5 | | – | – | 4 |
| | – | – | – | | – | – | – | | – | – | – | | – | – | 5 |

## 5.1.8 – 5.1.9 Queues

| Exit skills. Students should be able to[1]: |
|---|
| Describe the characteristics and applications of queues.<br>Construct algorithms using the access methods of queues.<br>Trace algorithms that use queues. |

*Characteristics of queues*

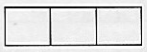
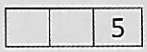
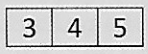
A queue stores a set of elements in a particular order and allows access only to the first item inserted. Items are retrieved in the order in which they are inserted. The queue is a First-In, First-Out (FIFO) data structure. The elements of a queue may be numbers, Boolean values, characters, objects, arrays, etc.

Queues utilize three methods:

1. **`enqueue()`**. Puts an item into the end of the queue.
2. **`dequeue()`**. Removes and returns the first item entered in the queue.
3. **`isEmpty()`**. Tests if a queue is empty. It will return true if queue contains no elements.

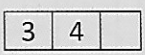
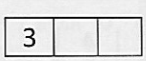
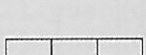
Suppose we want to add the elements 5, 4 and 3 in a queue named **`Numbers`**.

The following example presents this situation:

| | |
|---|---|
|  | **Queue initially empty** |
|  | **`Numbers.enqueue(5)`**<br>5 was added to the queue |
|  | **`Numbers.enqueue (4)`**<br>4 was added to the queue |
|  | **`Numbers.enqueue (3)`**<br>3 was added to the queue |

Suppose we want to remove all the elements from the queue.

The following example presents this situation:

| | |
|---|---|
|  | **Queue contains 3 numbers** |
|  | **`X = Numbers.dequeue()`**<br>First element was removed from the queue. This element was the number 5. 5 was assigned to variable **`X`**. |
|  | **`X = Numbers.dequeue()`**<br>First element was removed from the queue. This element was the number **4**. **4** was assigned to variable **`X`**. |
|  | **`X = Numbers.dequeue()`**<br>First element was removed from the queue. This element was the number 3. 3 was assigned to variable **`X`**. Queue is empty. |

*Applications of queues*

- Queues are used to model physical queues, such as people waiting in line at a supermarket checkout.
- The print queue displays the documents that are waiting to be printed. These documents will follow the first-send first-print policy.
- When sending data over the internet, various data packets wait in a queue to be sent.
- A server usually serves various requests. In most cases these requests are stored in a queue. The first-come first-served request procedure is followed.

*Algorithms that use queues*

**Programming Example 11: Use of a queue and arrays.**

A small school uses two buses to transport students. As soon as the buses arrive, all students enter a queue and a teacher uses a registry to check which students are present. The following algorithm uses two arrays to represent the school buses, a queue to represent the queue, and an array to represent the registry:

```
BUS1 = ["Roger", "John", "Nikos", "Marion", "Hellen"]
BUS2 = ["Nora", "Bill", "Eliza", "Takis", "Alex"]
REGISTRY = ["Alex", "John", "Elina", "Nikos", "Leo", "Marion",
"Hellen", "Nora", "Bill", "Eliza", "Takis", "Roger"]
STUDENTS = new Queue() //Queue for Students
A = ""
I = 0
FOUND = 1

//copy students from BUS1
loop I from 0 to 4
   STUDENTS.enqueue(BUS1[I])
end loop

//copy students from BUS2
loop I from 0 to 4
   STUDENTS.enqueue(BUS2[I])
end loop

loop while NOT(STUDENTS.isEmpty())
   A = STUDENTS.dequeue()
   loop I from 0 to 12
     if REGISTRY[I] = A then
       FOUND = 1
     end if
   end loop
     if FOUND = 1 then
       output A, "is not absent"
     end if
end loop
```

OUTPUT

```
Roger is not absent
John is not absent
Nikos is not absent
Marion is not absent
Hellen is not absent
Nora is not absent
Bill is not absent
Eliza is not absent
Takis is not absent
Alex is not absent
```

## Programming Example 12: Use of queues, arrays and a collection.

A supermarket has two express cashiers. The array **CASHIER1** contains the customers that enter the queue **CUSTOMER1**, while the array **CASHIER2** contains the customers that enter the queue **CUSTOMER2**. The **TIME** collection stores the names of the customers that waited more than 60 secs, counting from the moment that their turn to be served had come. The supermarket administration wishes to minimize the waiting for these two express cashiers. A questionnaire is sent by email, from the administration of the supermarket, to the customers stored in the collection **TIME** to understand why this situation took place. A message that outputs the overall slower express cashier is output at the end of the day.

```
CASHIER1 = ["Roger", "John", "Nikos", "Marion"]
CASHIER2 = ["Nora", "Bill", "Eliza", "Takis"]
CUSTOMER1 = new Queue()    //Queue for CUSTOMER1
CUSTOMER2 = new Queue()    //Queue for CUSTOMER2
TIME = new Collection()
A = ""
B = 0
C1 = 0
C2 = 0
I = 0
D1 = 0
D2 = 0
TOT_B = 0
TOT_C = 0
FOUND = 1

 //copy CUSTOMER1 from CASHIER1
loop I from 0 to 3
   CUSTOMER1.enqueue(CASHIER1[I])
end loop

 //copy CUSTOMER2 from CASHIER2
loop I from 0 to 3
   CUSTOMER2.enqueue(CASHIER2[I])
end loop

loop while NOT(CUSTOMER1.isEmpty())
 D1 = (CUSTOMER1.dequeue())
 C1 = Math.floor((Math.random() * 100) + 1)// use of random function
to generate random times between 1 sec to 100 sec
```

```
   if C1>60 then //only customers waiting more than 60 secs enter the
collection
      TIME.addItem(D1)
   end if
 TOT_B = TOT_B + C1
end loop

loop while NOT(CUSTOMER2.isEmpty())
 D2 = (CUSTOMER2.dequeue())
 C2 = Math.floor((Math.random() * 100) + 1)
   if C2>60 then
     TIME.addItem(D2)
   end if
 TOT_C = TOT_C + C2
end loop

TIME.resetNext()
loop while TIME.hasNext()
   output TIME.getNext()
end loop

if TOT_B > TOT_C then//outputs the slower cashier
   output "CASHIER1 is slower"
else
   output "CASHIER2 is slower"
end if
```

A POSSIBLE OUTPUT

```
Roger
John
Bill
Eliza
CASHIER2 is slower
```

### 5.1.10 Arrays as static stacks and queues

| Exit skills. Students should be able to[1]: |
| --- |
| Explain push and pop operations, and test on empty/full stack.<br>Explain enqueue and dequeue operations, and test on empty/full queue. |

*Algorithms to implement stacks using an array*

**Programming Example 13: Implementation of stack using an array.**

The program starts with an array of 10 elements. The methods used are the following:

**push()**

this method is used to add elements in the stack. Inserting an element increments **high** by 1 and adds the element in this array position. The **high** is incremented before the insertion of the new item takes place.

**pop()**

this method returns the value of the top element and then decrements **high**. It serves to remove the top element from the stack. The item removed actually remains in the array but is inaccessible.

**isempty()**

it is based on the **high** variable. It returns **true** (1) if the stack is empty.

**isfull()**

it is based on the **high** variable. It returns **true** (1) if the stack is full.

**size()**

it is based on the **high** variable. It returns the number of elements stored in the stack.

```
s_array = new Array()
s_array = [0,0,0,0,0,0,0,0,0,0]
maxsize = 10
high = -1
n = 0

pop()
push(1)
push(2)
push(7)
push(8)
pop()
push(9)
push(10)
push(9)
push(33)
push(29)
push(11)
push(49)
push(10)

output "---------"
output "high = ", high
output "---------"
output "s_array contains :", s_array
output "---------"
output "size of stack = ", size()

output
"////////////////////////////////////"
output "stack contents display and
removal"
output
"////////////////////////////////////"
loop while isempty() = 0
  n = pop()
  output n
end loop
```

**OUTPUT:**

Message: stack is empty
Message: stack is full
---
high = 9
---
s_array contains :
1,2,7,9,10,9,33,29,11,49
---
size of stack = 10
////////////////////////////////////
stack contents display and removal
////////////////////////////////////
49
11
29
33
9
10
9
7
2
1
////////////////////////////////////

**Explanation:**

This algorithm uses the s_array to implement a stack The methods used are **push()**, **pop()**, **isempty()**, **isfull()** and **size()**.

When this algorithm starts an array of

```
output
"//////////////////////////////////"


method push(n)
 if (isfull() == 1)  then
    output "Message: stack is full"
 else
   high = high + 1
   s_array[high] = n
 end if
end method

method pop()
 if (isempty() == 1) then
   output "Message: stack is empty"
 else
    high = high - 1
    return s_array[high+1]
  end if
end method

method isempty()
   if (high == -1) then
      return 1
   else
      return 0
   end if
end method

method isfull()
   if (high == maxsize-1) then
      return 1
   else
      return 0
   end if
end method

method size()
  return high+1
end method
```

ten elements is created.
**maxsize** variable is used to hold the maximum stack size, **high** variable is used to point the array position that is the top of the stack.

The first **pop()** instruction generates a "Message: stack is empty" output.

**push(1), push(2), push(7), push(8)** instructions add four elements in the stack.

**pop()** instruction removes 1 from the stack.

**push(9), push(10), push(9), push(33), push(29), push(11), push(49)** instructions add 7 elements in the stack. The stack is now full.

**push(10)** instruction causes "Message: stack is full " message to be displayed.

**Instruction "output "high = ", high"**
prints the number 9 which is the array position used to point the end of the queue.

Instruction **"output s_array contains :", s_array"** outputs the contents of the actual array used. The numbers 1,2,7,9,10,9,33,29,11,49 are printed.

The size of stack is 10

After a "stack contents display and removal" message a loop that removes and outputs all elements of stack is used. 49,11,29,33,9,10,9,7,2,1 are printed.

**Programming Example 14: Convert integer to binary using a stack.**

```
//This algorithm uses a stack to convert an integer to its binary
equivalent

//Declaration of variables
s_array = new Array()
s2_array = new Array()
s_array = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
maxsize = 10
high = -1
x = 0
y = 0
n = 0
t = 0
dig_value = 0
number = 123

output "Convert number ", number

//Call method convert_in_binary
convert_to_binary(number) //max number is 1023

//Use of an auxiliary array to properly output the result
output " "
output "Final result"
loop a from 0 to 9
  s2_array[a] = s_array[9-a]
end loop
output s2_array

method convert_to_binary(x)
 output "Calculations"
 loop while x > 0
   y = x mod 2
   push(y) //use of push method
   x = div(x,2) //division of x over 2
 end loop
 //the next loop will use the isempty method
 loop while isempty() = 0
   t = pop() //use of pop method
   dig_value = Math.pow(2, (high+1)) //2^(high+1)
   output "Binary digit number", high+1,"(",dig_value,")", "is", t
 end loop
end method

method push(n)
 if (isfull() == 1)  then
   output "Message: stack is full"
 else
   high = high + 1
   s_array[high] = n
 end if
end method
```

```
method pop()
 if (isempty() == 1) then
   output "Message: stack is empty"
 else
    high = high - 1
    return s_array[high+1]
  end if
end method

method isempty()
   if (high == -1) then
      return 1
   else
      return 0
   end if
end method

method isfull()
   if (high == maxsize-1) then
      return 1
   else
      return 0
   end if
end method
```

OUTPUT

```
Calculations
Binary digit number 6 ( 64 ) is 1
Binary digit number 5 ( 32 ) is 1
Binary digit number 4 ( 16 ) is 1
Binary digit number 3 ( 8 ) is 1
Binary digit number 2 ( 4 ) is 0
Binary digit number 1 ( 2 ) is 1
Binary digit number 0 ( 1 ) is 1

Final result
0,0,0,1,1,1,1,0,1,1
```

*Algorithms to implement queues using an array*

**Programming Example 15: Implementation of queue using an array.**

| INDEX | *0* | *1* | *2* | *3* | *4* | *5* | *6* |
|---|---|---|---|---|---|---|---|
| | 5 | 3 | 9 | 7 | 8 | | |
| | **FRONT** | | | | | **REAR** | |

When using an array to implement a queue, insertion takes place at the **REAR** index, while deletion takes place at the **FRONT** index only. At the beginning both **FRONT** and **REAR** are 0. When entering the first element, **FRONT** remains 0, while **REAR** becomes 1. When entering another element, **FRONT** again remains 0, while **REAR** becomes 2. When entering yet another element, **FRONT** remains 0 and **REAR** becomes 3. If we remove an element, **FRONT** becomes 1 and **REAR** remains 3. If we remove another element, **FRONT** becomes 2 and **REAR** remains 3. If we remove yet another element, both **FRONT** and **REAR** become 0, since the queue is empty.

The following algorithm implements this approach. Unfortunately, this array-based implementation is tricky. It works well when entering elements and then removing them all before entering new elements again. This is not the case when adding and deleting data in a random order since the end of the array will eventually be reached and an out-of-bounds exception will be raised.

```
q_array = new Array()
q_array = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
FRONT = 0
REAR = 0
SIZE = 10
n = 0
dequeue()
enqueue(71)
enqueue(1)
enqueue(2)
enqueue(112)
enqueue(14)
enqueue(52)
enqueue(67)
enqueue(14)
enqueue(52)
enqueue(62)
dequeue()
dequeue()
dequeue()
dequeue()
dequeue()
dequeue()
enqueue(61)

output "Queue contents display"
 output "---------"
 if (FRONT == REAR) then
    output "Message: queue is empty"
 else
   loop I from FRONT to REAR-1
     n = q_array[I]
     output n
   end loop
 end if
 output "---------"
```

```
method enqueue(N)
 if REAR == SIZE then
    output "Message: queue is full"
 else
    q_array[REAR] = N
    REAR = REAR + 1
 end if
end method

method dequeue()
 if FRONT == REAR then
   output "Message: queue is empty"
 else
    N = q_array[FRONT]
    if (FRONT+1 == REAR) then
      REAR = 0
      FRONT = 0
    else
      FRONT = FRONT + 1
    end if
 end if
end method
```

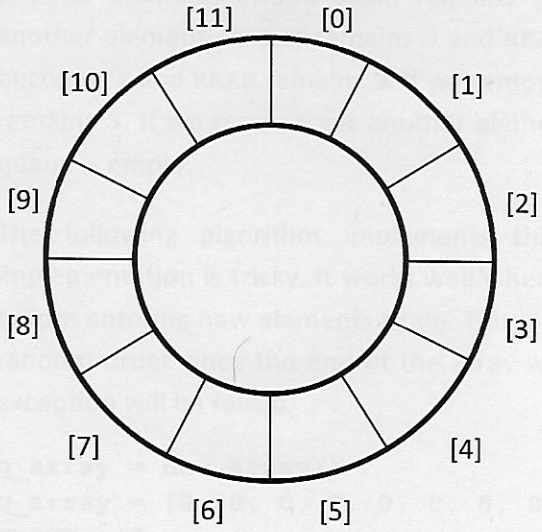
OUTPUT

```
Message: queue is empty
Message: queue is full
Queue contents display
---------
67
14
52
62
---------
```

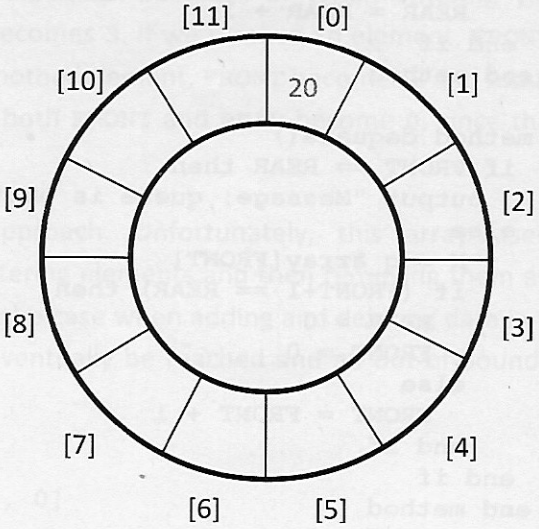
As we can see the queue contains only 4 elements. Although the array can hold 10 elements the **FRONT** is now 7 and the **REAR** is **10** so **enqueue(61)** will generate the **queue is full** message. This situation can be solved by using a circular implementation of a queue.

*Algorithms to implement a circular queue using an array*

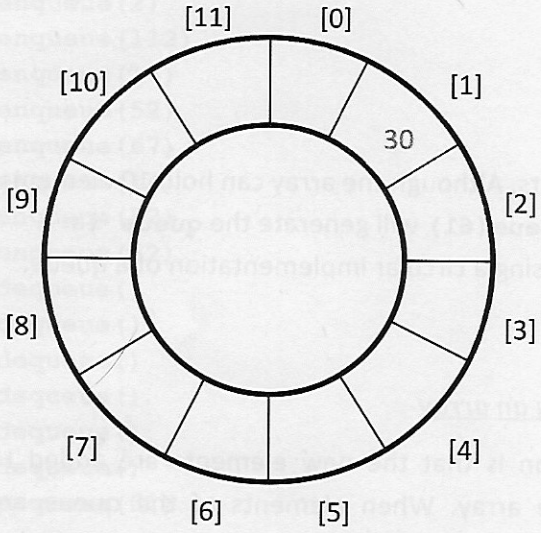
The problem with the previous implementation is that the new elements are added to successively higher-numbered positions in the array. When elements of the queue are deleted, the **FRONT** index increases and this process continues until the queue runs out of space. The array might have free positions at the indices that are smaller than the **FRONT** index, but these positions are unusable. The following circular implementation of a queue solves this problem:



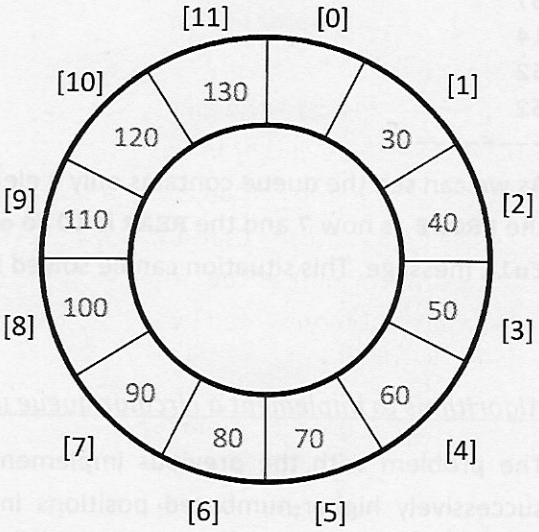

FRONT = 0
REAR = -1



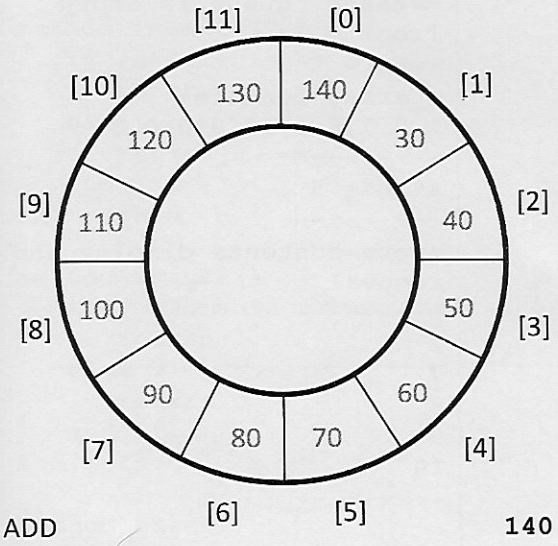

ADD 20
FRONT = 0
REAR = 0




DELETE 20
ADD 30
FRONT = 1
REAR = 1




ADD
30,40,50,60,70,80,90,100,110,120,130
FRONT = 1
REAR = 11




ADD **140**
FRONT = **1**
REAR = **0**
This is the benefit of a circular queue.
Item **140** was inserted in the array index **0**.

**Figure 5.3: Explanation of the operation of a circular queue**

The following algorithm starts with an array of **10** elements. The methods used are the following:

**enqueue()**
This method is used to add elements to the queue. Inserting an element increments **rear** by **1** and inserts the element in the new array position where **rear** points to. If **rear** is at the end (top) of the array, then **rear** should be set to **-1** before the addition of the element takes place. This means that a wraparound takes place and the next element will be placed at the start (bottom) of the array. Finally, the variable that holds the number of elements, **nelements**, is incremented by **1**.

**dequeue()**
This method is used to remove elements from the queue. A temporary variable, **temp**, is used to hold the value of **front**. **front** is then incremented by **1**. If **front** equals to the array length, then a wraparound takes place and **0** is assigned to **front**. Finally, the variable that holds the number of elements, **nelements** is decremented by **1**.

**isempty()**
This is based on the **nelements** variable. It returns **true** (1) if the queue is empty.

**isfull()**
This is based on the **nelements** variable. It returns **true** (1) if the queue is full.

**size()**
This is based on the **nelements** variable. It returns the number of elements stored in the queue.

```
q_array = new Array()
q_array = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 //you can replace the previous two
 lines
 //with q_array = new Array(10)
maxsize = 10
front = 0
rear = -1
nelements = 0
n = 0
dequeue()
enqueue(1)
enqueue(2)
enqueue(7)
enqueue(8)
dequeue()
enqueue(9)
enqueue(10)

output "front = ", front
output "rear = ", rear
output "q_array contains :", q_array
output "---------"
output "size = ", size()
output "---------"

output "queue contents display and
removal"
output "---------"
loop while isempty() = 0
  n = dequeue()
  output n
end loop
output "---------"


method enqueue(n)
 if isfull() = 1  then
    output "Message: queue is full"
 else
   if (rear == maxsize-1) then
      rear = -1
   end if
     rear = rear +1
     q_array[rear] = n
     nelements = nelements + 1
  end if
end method

method dequeue()
 if isempty() = 1 then
   output "Message: queue is empty"
 else
    temp = q_array[front]
    front = front + 1
    if (front==maxsize) then
      front = 0
    end if
    nelements = nelements - 1
    return temp
  end if
```

**Output:**

```
Message: queue is empty
front = 1
rear = 5
q_array contains :
1,2,7,8,9,10,0,0,0,0
---------
size = 5
---------
queue contents display and
removal
---------
2
7
8
9
10
---------
```

**Explanation:**

This algorithm uses the `q_array` to implement a circular queue. The methods used are: `enqueue()`, `dequeue()`, `isempty()`, `isfull()` and `size()`.

When this algorithm begins, an array of ten elements is created. `maxsize` variable is used to hold the maximum queue size, the `front` variable is used to point to the start of the queue, the `rear` variable is used to point to the end of the queue and `nelements` is used to hold the total number of elements stored in the queue.

The first `dequeue()` instruction generates a "`Message: queue is empty`" output.

`enqueue(1)`, `enqueue(2)`, `enqueue(7)`, `enqueue(8)` instructions add four elements in the queue.

`dequeue()` instruction removes 1 from the queue.

`enqueue(9)` and `enqueue(10)` add two elements to the queue.

Instruction `output "front = ", front` prints the number 1

```
end method

method isempty()
   if (nelements == 0) then
      return 1
   else
      return 0
   end if
end method

method isfull()
   if (nelements == maxsize) then
      return 1
   else
      return 0
   end if
end method

method size()
  return nelements
end method
```

which is the array position used to point to the front of the queue.

Instruction **`output "rear = ", rear`** prints the number **5** which is the array position used to point to the end of the queue.

Instruction **`output "size = ", size()`** outputs **`size = 5`**, which is the size of the queue.

Instruction output **`"q_array contains:", q_array`** outputs the contents of the actual array used. The numbers **1,2,7,8,9,10,0,0,0,0** are printed.

After a "**`queue contents display and removal`**" message, a loop that removes and outputs all elements of queue is used. **`2 7 8 9 10`** are printed.

### 5.1.11 Features and characteristics of a dynamic data structure

| Exit skills. Students should be able to[1]: |
| --- |
| Describe the features and characteristics of a dynamic data structure.<br>Understand nodes and pointers. |

*Linked lists will be examined at the level of diagrams and descriptions. Students are not expected to construct linked list algorithms using pseudocode.*

Suppose you want to develop a program to handle a variable number of airplanes that arrive at an airport every day. The array based implementation has the disadvantage that its size must be predetermined. The size of the array must be determined before the actual use of the array. Declaring a large array means allocating an amount of memory that might be not

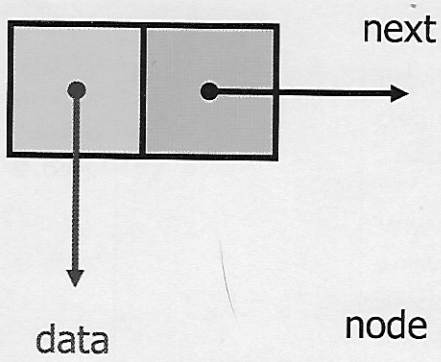



**Figure 5.4: A node of a linked list**

utilized, while declaring a small array means that the array may run out of space during the execution of the program. The solution to this inconvenient situation is to use dynamic allocation of memory. The resulting data structure will be flexible enough to shrink or grow, as the request for data storage decreases or increases respectively. This provides the programmer with the ability to control the amount of memory that is utilized.

In programming, a **node** is a basic unit (object) that contains both data and a **pointer**. A pointer is a field of the node whose value points to another object, stored in some other memory location. Each node in a linked list stores a pointer to the next value of the linked list. Thus, every node requires memory for both its data and its pointer. THE NULL pointer is a special pointer that points to nothing, meaning that it has no **pointee**. The NULL pointer is drawn as a diagonal line between the left lower corner and the right upper corner of the pointers variable box.

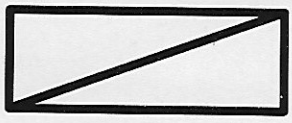


**Figure 5.5: The NULL pointer**

### 5.1.12 Operation of linked lists

| Exit skills. Students should be able to[1]: |
| --- |
| Describe how linked lists operate logically. |

A linked list is constructed from a series of nodes. Every node of the list is a distinct object that contains both data, as well as a reference (pointer) to the next node. A reference refers to an object's address in the RAM. In C++ we use pointers, while in Java we use references.

A linked list is very different from an array. Although they are both used to create lists, their operational characteristics are completely different. In an array, each element resides in a particular position which can be directly accessed using an index. In a linked list, a particular element can only be accessed by following the references (pointers) of all the previous element. In a way, linked lists resemble a chain. There is no way to access an element directly. One can only access every element in turn, starting with the initial one in the list.

It is important to stress the difference between the **logical representation**, which refers to how the data and the links are "seen" by the programmer, and the **physical representation**, which refers to the underlying mechanisms that store the data in RAM. The physical representation includes details like memory addresses, type of data, number of bytes used, the way the pointers are handled etc. This is another example of abstraction where all unnecessary details are hidden.

**Key Characteristics of a linked list:**

- To traverse a linked list, you start at the first node and then go from node to node, following each node's pointer to find the next node.
- A node with a specific key value can be found by traversing the list. Once found, a node's data can be accessed.
- A linked list consists of a sequence of nodes.
- Each node contains data and a pointer.
- A linked list may be empty.
- The length of a linked list is the number of elements that it contains.
- The last node contains a null pointer.
- A node's successor is the next node.
- A node's predecessor is the previous node.

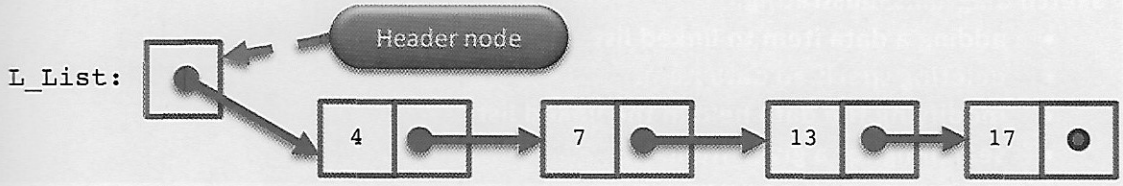



**Figure 5.6: A diagram of linked list with four elements**

In the previous Figure the existence of a dummy header node, which is just an initial node that exists at the front of a linked list even when the list is empty, is assumed. Its purpose is to point at the first element and to keep the linked list from being null. This node doesn't contain any data, but only a pointer.

*Representing linked lists with pseudocode*

Suppose we have an object named **student** with the following properties:

```
Name (of type) string
Surname (of type) string
ID (of type) integer
```

A node object will have one additional field to store information about the reference to the next node of the linked list:

```
Next (of type) pointer (or reference)
```

So the node of a single linked list that is used to store **student** objects should have the following fields:

```
Name (of type) string
Surname (of type) string
ID (of type) integer
Next (of type) pointer (or reference)//reference to the next node
```

It is clear that the node is not only the data that it stores, but it also includes a pointer.

Another approach would be to create two classes. A **student** class and a **node** class. In this approach the data are not placed directly in the node. A reference is used in the node object to represent the student's data.

| node | Student |
|---|---|
| Student s1 //reference to student object Next of type pointer (or reference) | Name of type string<br>Surname of type string<br>ID of type integer |

**Table 5.3: Pseudocode for the two classes**

It is important to mention that the **ID** will be used as the key value. So all linked list operations (finding, inserting, deleting etc.) will be implemented according to this key.

### 5.1.13 Sketch linked lists

| Exit skills. Students should be able to[1]: |
|---|
| Sketch linked lists (single, double and circular).<br>Sketch diagrams illustrating:<br>• adding a data item to linked list.<br>• deleting specified data item.<br>• modifying the data held in the linked list.<br>• searching for a given item. |

### Empty linked list

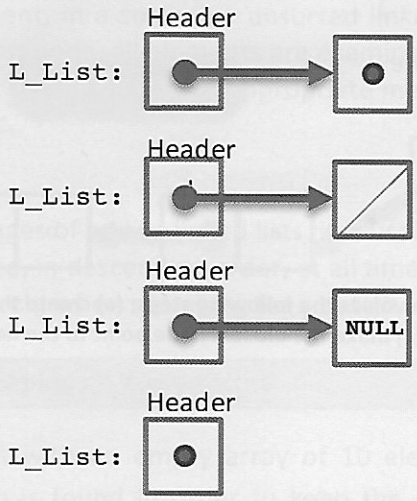



**Figure 5.7: Possible representations of an empty linked list (`L_List`).**

### Adding (inserting)

*After an element*

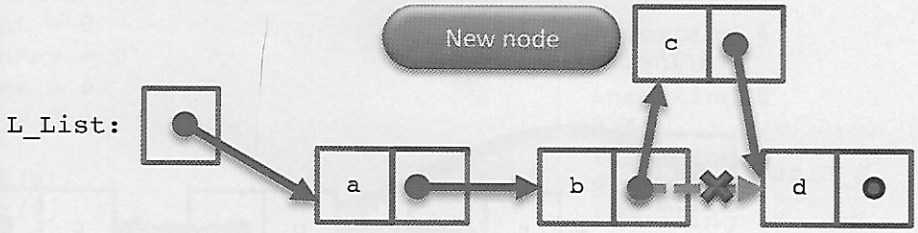



**Figure 5.8: Adding an element involves 4 steps: (a) find the node you want to insert after, (b) create the new node, (c) copy the pointer from the node that's already in the list, (d) change the pointer in the node that's already in the list to point to the inserted node.**

*At the beginning*

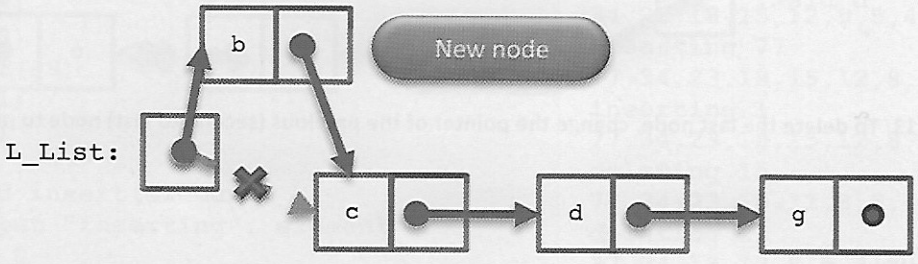



**Figure 5.9: Inserting to the head involves the following steps: (a) create the new node, (b) make the new node point to the old first, (c) make header node point to the inserted node.**

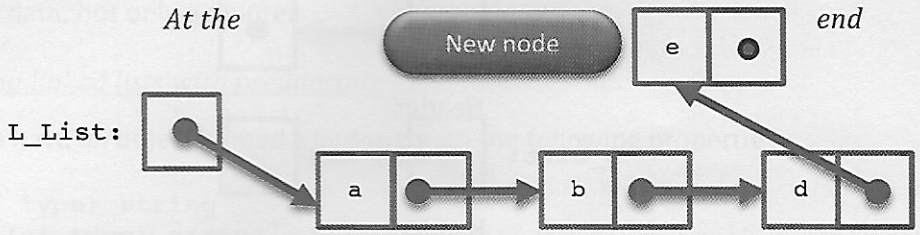



**Figure 5.10: Inserting to the tail involves the following steps: (a) create the new node, (b) make the new node point to null, (c) make the old last node point to the new inserted node.**

### *Deleting*

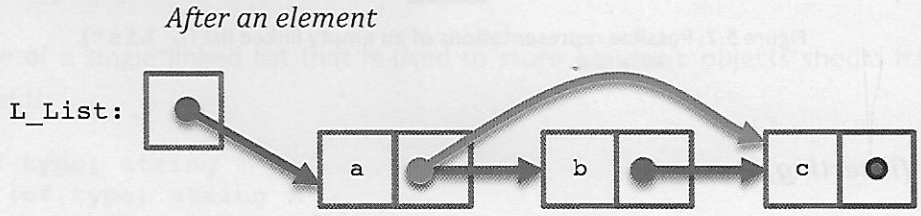



**Figure 5.11: To delete a node, just change the link in its previous node to point its next node.**

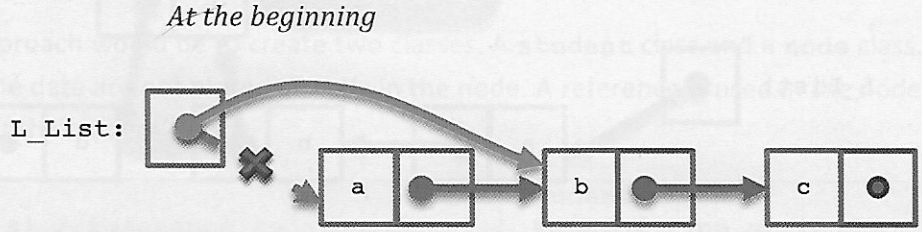



**Figure 5.12: To delete the first node, change the link in the header to point to the next (second) node.**

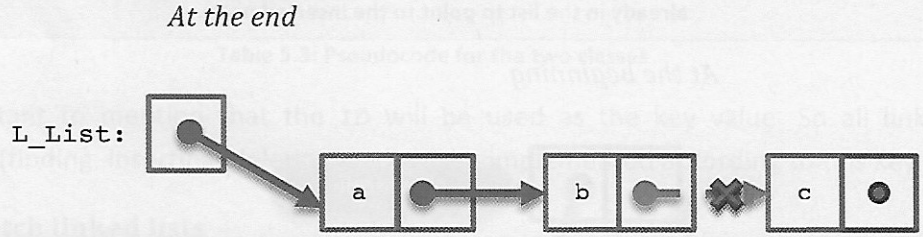



**Figure 5.13: To delete the last node, change the pointer of the previous (second to last) node to point to NULL.**

*Modifying*

To modify the data of a node one can follow two different procedures:

1. Delete the node and insert a new node with different data in the same position of the list
2. Find the node and change its data by replacing it with the new data

*Searching*

To search for a specific element, in a sorted or unsorted linked list, linear search must be followed. Starting from the first node, all elements are examined until the desired element is found. If the element is not in the linked list, an appropriate message is returned.

*Sorted linked list*

Before discussing the advantages of sorted linked lists, it is useful to examine a program that keeps all array elements sorted, in descending order, at all times. Keeping a list sorted makes it possible to apply binary search, when searching for a data item.

**Programming Example 16: Keeping an array sorted.**

The following program starts with an empty array of 10 elements. When an element is inserted, the correct location is found in order to keep the array sorted in a descending order. Deletion works by shifting elements, with higher index numbers, to fill in the gap left by the deletion process. To find the position of an element to be deleted, binary search is applied. The advantage of using an ordered array becomes apparent when binary search is applied, since it performs much faster than a sequential search.

```
ARRAY = new Array(10)
n_elements = 0
element = 0
searchKey = 0
C_index = 0
found = 0

insert(4)
insert(8)
insert(5)
insert(12)
insert(34)
insert(18)
insert(15)
insert(23)
insert(77)
insert(1)
del(15)
del(23)
insert(99)
del(11)
del(4)

method insert(element)
  output "inserting", element
  j = 0
  k = 0
  loop while (j < n_elements
              AND ARRAY[j]>element)
    j = j+1
  end loop
  k = n_elements
  loop while k > j
    ARRAY[k] = ARRAY[k-1]
```

OUTPUT

```
inserting 4
4,,,,,,,,,
inserting 8
8,4,,,,,,,,
inserting 5
8,5,4,,,,,,,
inserting 12
12,8,5,4,,,,,,
inserting 34
34,12,8,5,4,,,,,
inserting 18
34,18,12,8,5,4,,,,
inserting 15
34,18,15,12,8,5,4,,,
inserting 23
34,23,18,15,12,8,5,4,,
inserting 77
77,34,23,18,15,12,8,5,4,
inserting 1
77,34,23,18,15,12,8,5,4,1
deleting 15
77,34,23,18,12,8,5,4,1,
deleting 23
77,34,18,12,8,5,4,1,,
inserting 99
99,77,34,18,12,8,5,4,1,
deleting 11
11 not found
deleting 4
99,77,34,18,12,8,5,1,,
```

```
    k = k - 1
  end loop
  ARRAY[j] = element
  n_elements = n_elements + 1
  output ARRAY
end method

method del(searchKey)
  output "deleting", searchKey
  LOW = 0
  HIGH = n_elements - 1
  C_index = 0
  m = 0
  found = 0
  loop while (LOW <= HIGH)
    C_index = div ((LOW + HIGH), 2)
    if (ARRAY[C_index] == searchKey)
    then
      found = 1
      loop m from C_index to
n_elements - 2
        ARRAY[m] = ARRAY[m+1]
      end loop
      ARRAY[n_elements - 1] = ""
      n_elements = n_elements - 1
      output ARRAY
    else if (ARRAY[C_index] >
searchKey) then
      LOW = C_index + 1
    else
      HIGH = C_index - 1
    end if
  end loop
  if (found = 0) then
    output searchKey, "not found"
  end if
end method
```

Sometimes, it is very useful to maintain a sorted linked list. In such a linked list, all data are stored according to a key value. A programmer can use a sorted linked list in the same way that a sorted array is used. Although a linked list is more difficult to implement than a sorted array, the advantages of a sorted linked list is that the elements do not need to be moved, but only pointers need to be altered. This results in a high speed element insertion, which, in addition to the fact that the linked list can easily expand to any size that is supported by the available RAM, makes linked lists ideal in various situations. To insert an element in a sorted linked list, one must first search through the linked list until he/she finds the correct place to insert the new element. After that, the element can be inserted in the usual manner described above.

*Double-linked lists*

In a double-linked list each node contains data, a pointer to its successor, *and* a pointer to its predecessor. The header node points to the first node *and* to the last node of the list. If the linked list is empty, the pointers point to NULL. The first node and the last node of such a list are directly accessible without traversal, and allow traversal of the list from the beginning or

the end. The pointers of each node allow traversal of the list in either direction. Another advantage of a double-linked list is that deletion and insertion before a node, become easier. A serious disadvantage of the double-linked list is the additional space used. The double-linked list requires two pointers per node, and so it needs twice as much overhead as the singly linked list. Applications of double-linked lists include web browsers, where the back and forward buttons are used for backward and forward navigation, implementation of undo and redo functions etc.

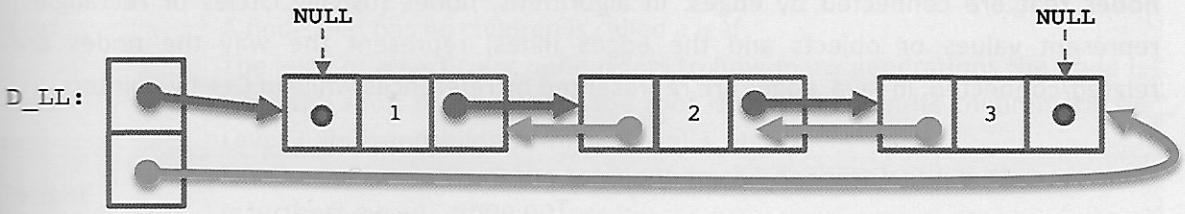



**Figure 5.14: A double-linked list**

*Circular-linked lists*

A circular list is a linked list in which the last link points back to the first link. In such a list it is easy to loop and access all nodes circularly, and one has the ability to traverse the entire list starting from any node. The implementation of the circular-linked list is more complex and extra caution is needed so as not to end up in an infinite loop. Applications of Circular-linked lists include OS time sharing algorithms and multiplayer games.

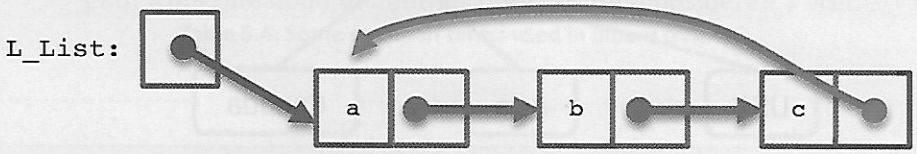



**Figure 5.15: A Circular-linked list with a header node**

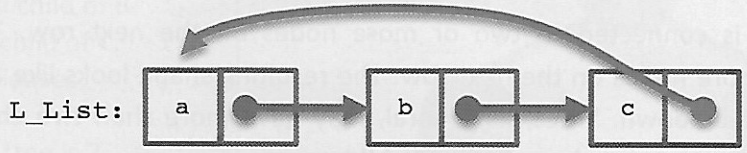



**Figure 5.16: A circular list without a header node**

## Trees

*Binary trees will be examined at the level of diagrams and descriptions. Students are not expected to construct tree algorithms using pseudocode. Tracing and constructing algorithms are not expected.*

### 5.1.14 Logical operation of trees

| Exit skills. Students should be able to[1]: |
| --- |
| Describe how binary and non-binary trees operate logically. |

**Trees** combine the quick insertion and deletion of linked lists, as well as the quick searching of an ordered array. Tress in general are fascinating dynamic data structures that have been studied as abstract mathematical entities. They belong to the **graph** category and consist of **nodes** that are connected by **edges**. In algorithms, nodes (usually circles or rectangles) represent values or objects and the edges (lines) represent the way the nodes are related/connected. In Java, edges are represented by references while in C++ by pointers.

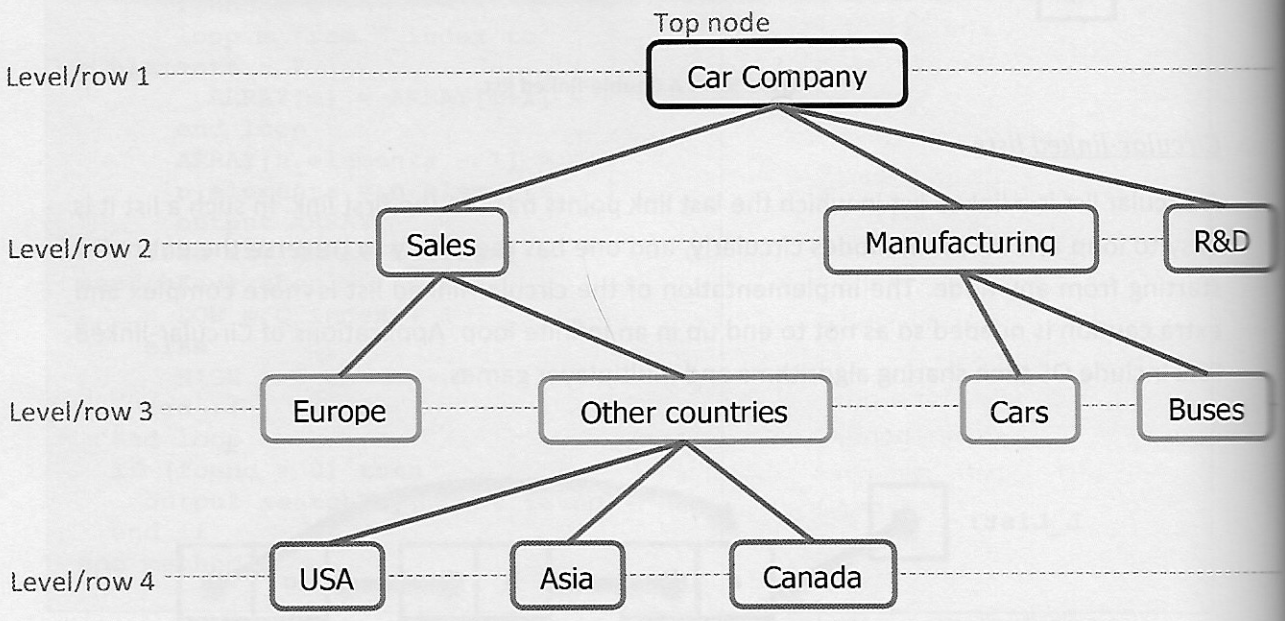



**Figure 5.17: A tree used to represent an abstract model of a hierarchical structure.**

The top node is connected to two or more nodes on the next row. These nodes are connected to more nodes on the next row. The resulting shape looks like a "real tree" that was turned upside-down. Trees, in general, may have more than two children per node. However, **a binary tree may have 0, 1 or 2 children.**

### 5.1.15 Binary-tree related terminology

| Exit skills. Students should be able to[1]: |
|---|
| Define the terms: parent, left-child, right-child, subtree, root and leaf. |

| Term | Explanation |
|---|---|
| **Children-child** | The nodes below a given node are its children nodes. |
| **Key** | It is a data field of a node, which may be used to search for the node or perform other operations on it. |
| **Leaf** | A node that has no children is called a leaf. |
| **Level** | The level of a particular node refers to how many generations the node is from the root. If we assume the root is Level 1, then its children will be Level 2, its grandchildren will be Level 3, etc. |
| **Height** | Number of edges from the top node to the deepest leaf (i.e. the one that is furthest away). |
| **Parent** | All nodes except from the root, which has no parent node, have exactly one edge running upward to their parent node. |
| **Path** | Suppose one wants to travel, from node to node, along the edges that link them. The sequence of nodes that are travelled is called a path. |
| **Root** | The node at the top of the tree is called the root. |
| **Subtree** | Any node may be considered to be the root of a subtree. That subtree will consist of the node's descendants. |
| **Traversing** | To traverse means to visit all the nodes of the tree in some specified order. |
| **Visiting** | To visit a node means to arrive at a node for the purpose of performing some operation on the node. If an algorithm passes over a node on the path from one node to another then it is not considered a visit. |

**Table 5.4: Some common terms used in binary tree**

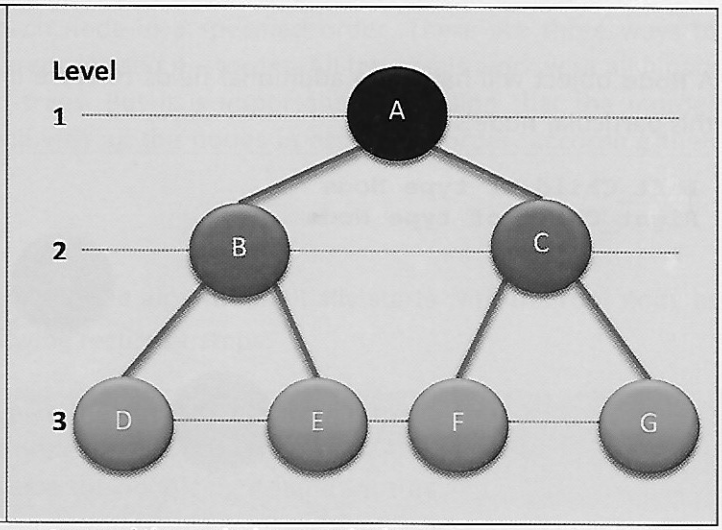
| | |
|---|---|
| A is the root node.<br>B is the parent of D and E.<br>F and G are the children of C.<br>E is the right-hand child of B.<br>F is the left-hand child of C.<br>D, E, F, and G are leaves.<br>The level of E is 3.<br>The height of the tree is 2.<br>The ancestors of node E is A and B.<br>The descendants of node C is F, and G.<br>B is the root node of the subtree consisted of B and its descendants, D and E. |  |

**Figure 5.18: A binary tree and the equivelant terminology**

**A tree that has at most two children is called a binary tree.**

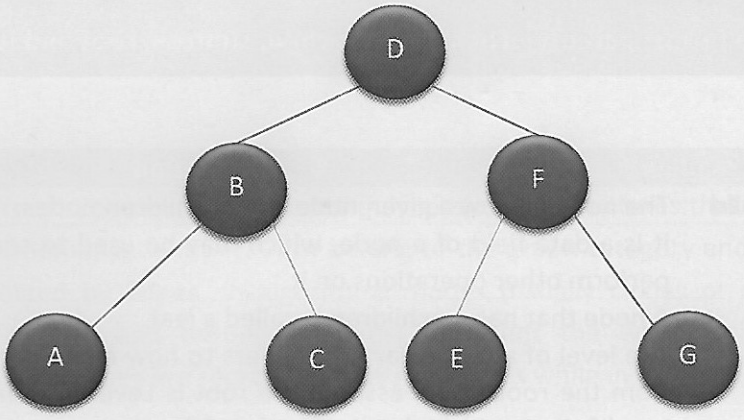



**Figure 5.19: A binary search tree**

A recursive definition of a binary tree is the following: Every binary tree has only one root. Each node of the tree can be considered as the root of a subtree of the tree. Consequently, every tree consists of a root and one or more subtrees. Each subtree is a tree.
In a binary tree when a node's left-hand child has a key less than its parent and a node's right-hand child has a key greater than or equal to its parent, then the tree is a **binary search tree**. These trees keep their keys in sorted order and allow fast lookup of data. The binary search algorithm can be applied when looking for a particular key in the tree.

*Representing a binary tree using pseudocode*

Suppose we have an object named **`Student`** with the following properties:

```
Name of type string
Surname of type string
ID of type integer
```

A **`Node`** object will have two additional fields to store information about the references to this particular node's children:

```
Left_Child of type Node
Right_Child of type Node
```

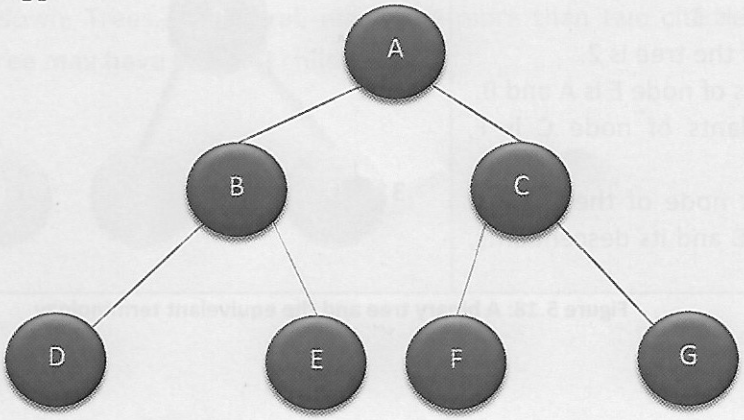



**Figure 5.20: A binary tree that it is not a binary search tree**

So the **Node** of a tree that is used to store **Student** objects should have the following fields:

```
Name of type String
Surname of type String
ID of type Integer
Left_Child of type Node //reference to Node's left-hand child
Right_Child of type Node //reference to Node's right-hand child
```

It is clear that the **Node** encompasses more than just the data that it stores (i.e.: the **Name**, **Surname** and **ID**). It may also include references to its children, if any.

Another approach would be to create two separate classes: a **Student** and a **Node** class. In this approach the data are not placed directly in the **Node**. A reference is used in the **Node** object to represent the student's data.

| Node object | Student object |
|---|---|
| Student s1 //reference to student object<br>Left_Child of type node<br>Right_Child of type node | Name of type string<br>Surname of type string<br>ID of type integer |

**Table 5.5: Pseudocode for the two classes**

It is important to mention that the **ID** will be used as the key value. As such, all tree operations (finding, inserting, deleting etc.) will be implemented according to this key.

### 5.1.16 Tree traversal

| Exit skills. Students should be able to[1]: |
|---|
| State the result of inorder, preorder and postorder tree traversal. |

**Traversing a tree** means to visit each node in a specified order. There are three ways to implement tree traversal: inorder, preorder and postorder. All traversals work with all binary trees, not only with binary search trees. But it is important to mention that the inorder traversal of a binary search tree will visit all the nodes in ascending order, according their key values.

*Inorder traversal algorithm*

It is assumed that the tree is not empty. The algorithm initially starts with the root node as an argument and performs the following recursive steps:

1. The algorithm calls itself to traverse the node's left-hand subtree.
2. The algorithm visits the current node.
3. The algorithm calls itself to traverse the node's right-hand subtree.

In pseudocode this algorithm looks like:

```
in_Order(localRoot)

if(localRoot != null) then
```

```
  in_Order(localRoot.Left_Child)
  output localRoot.ID //ID is the key value
  in_Order(localRoot.Right_Child)
end if
```

This recursive method is first called with the root of the tree as an argument, as such:

```
in_Order(root)
```

*Preorder traversal*

It is assumed that the tree is not empty. The algorithm initially starts with the root node as an argument and performs the following recursive steps:

1. The algorithm visits the current node.
2. The algorithm calls itself to traverse the node's left-hand subtree.
3. The algorithm calls itself to traverse the node's right-hand subtree.

In pseudocode this algorithm looks like:

```
pre_Order(localRoot)

if(localRoot != null) then
  output localRoot.ID //ID is the key value
  pre_Order(localRoot.Left_Child)
  pre_Order(localRoot.Right_Child)
end if
```

This recursive method is first called with the root of the tree as an argument, as such:

```
pre_Order(root)
```

*Postorder traversal*

It is assumed that the tree is not empty. The algorithm initially starts with the root node as an argument and performs the following recursive steps:

1. The algorithm calls itself to traverse the node's left-hand subtree
2. The algorithm calls itself to traverse the node's right-hand subtree
3. Visit the node //Recall that visit a node means to perform an action

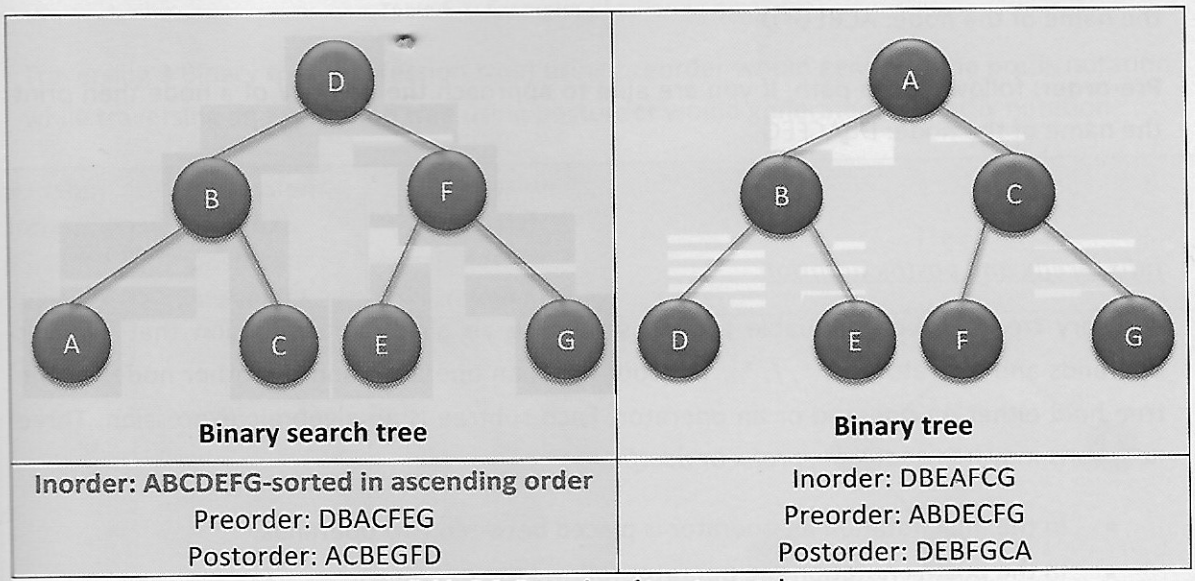
In pseudocode this algorithm looks like:

```
post_Order(localRoot)

if(localRoot != null) then
  post_Order(localRoot.Left_Child)
  post_Order(localRoot.Right_Child)
  output localRoot.ID //ID is the key value
end if
```

This recursive method is first called with the root of the tree as an argument, as such:

```
post_Order(root)
```




| Binary search tree | Binary tree |
|---|---|
| **Inorder: ABCDEFG-sorted in ascending order**<br>Preorder: DBACFEG<br>Postorder: ACBEGFD | Inorder: DBEAFCG<br>Preorder: ABDECFG<br>Postorder: DEBFGCA |

**Table 5.6: Examples of tree traversals**

*Tree traversal – a practical approach*

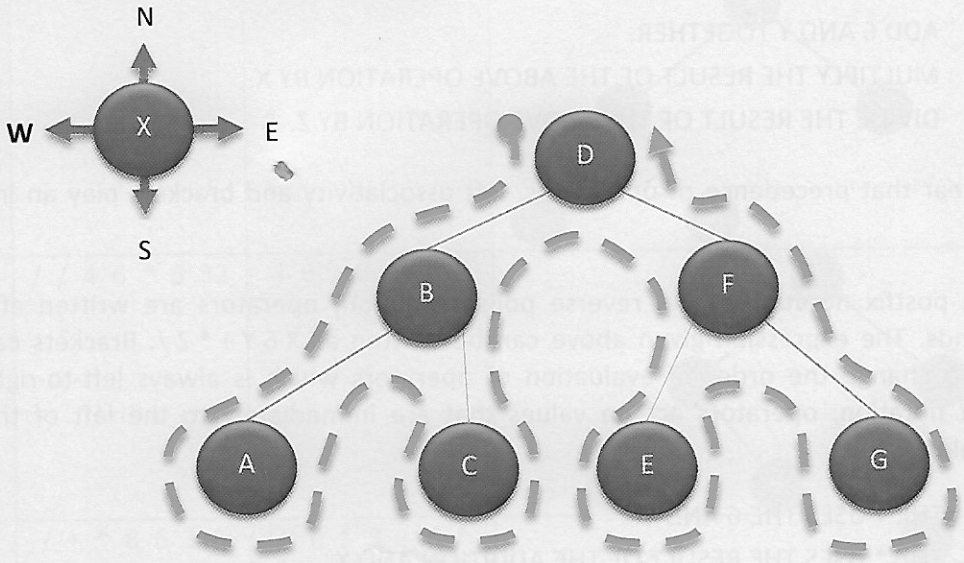



**Figure 5.21: Diagram that illustrates empirical tree traversal**

The approach presented in Figure 5.21 is very useful and can be used during examination to answer tree traversal questions. Suppose every node is illustrated as a circle and has four points at 0, 90, 180 and 270 degrees. The names of these points are N for 0°, E for 90°, S for 180° and W for 270° respectively. Figure 5.21 illustrates an orange coloured path that circles the tree starting from the root. The path tightly follows the outline of the tree without intersecting with any path or node.

**In-order traversal:** following the path, if you are able to approach the point S of a node then print the name of the node: ABCDEFG

**Post-order:** following the path, if you are able to approach the point E of a node then print the name of the node: ACBEGFD

**Pre-order:** following the path, if you are able to approach the point W of a node then print the name of the node: DBACFEG

*Infix, prefix and postfix notation*

A binary tree can be a valuable tool to symbolize an algebraic expression that involves operands and operators (+, -, /, *). The root holds an operator, and the other nodes of the tree hold either an operand or an operator. Each subtree is an algebraic expression. Three notations may be used: *infix*, *prefix* or *postfix*.

- In the infix notation an operator is placed between two operands.
- In the postfix notation the operator follows the operands.
- In the prefix notation the operator comes before the operands.

In the infix notation an algebraic expression such as X*(6+Y)/Z means:

- ADD 6 AND Y TOGETHER.
- MULTIPLY THE RESULT OF THE ABOVE OPERATION BY X.
- DIVIDE THE RESULT OF THE ABOVE OPERATION BY Z.

It is clear that precedence of operations, left associativity and brackets play an important role.

In the postfix notation (a.k.a. reverse polish notation) operators are written after their operands. The expression given above can be written as X 6 Y + * Z /. Brackets cannot be used to change the order of evaluation of operators which is always left-to-right. So in postfix notation, operators act on values that are immediately to the left of them. For example:

- THE + USES THE 6 AND Y.
- THE * USES THE RESULT OF THE ADDITION AND X.
- THE / USES THE RESULT OF THE MULTIPLICATION AND Z.

In the prefix notation (a.k.a. polish notation) operators are written before their operands. Operators are evaluated from left-to-right and brackets are unnecessary since operators act on the two nearest values to the right. The expressions given above can be written as: / * X + 6 Y Z. In this example:

- THE + USES 6 AND Y.
- THE * USES THE RESULT OF THE ADDITION AND X.
- THE / USES THE RESULT OF THE MULTIPLICATION AND THE Z.

| Infix | Postfix | Prefix |
|---|---|---|
| X * Y + Z / 2 | X Y * Z 2 / + | + * X Y / Z 2 |
| X * (Y + Z) / 2 | X Y Z + * 2 / | / * X + Y Z 2 |
| X * (Y + Z / 2) | X Y Z 2 / + * | * X + Y / Z 2 |

**Table 5.7: Examples of different notations**

Traversing a binary tree (expression tree) using preorder would generate the prefix notation while traversing an expression tree using postorder would generate the postfix notation.

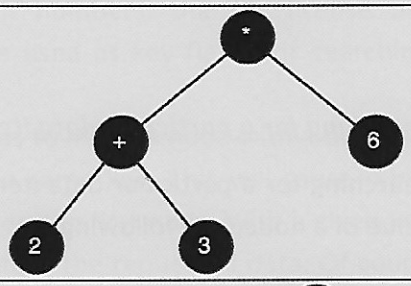
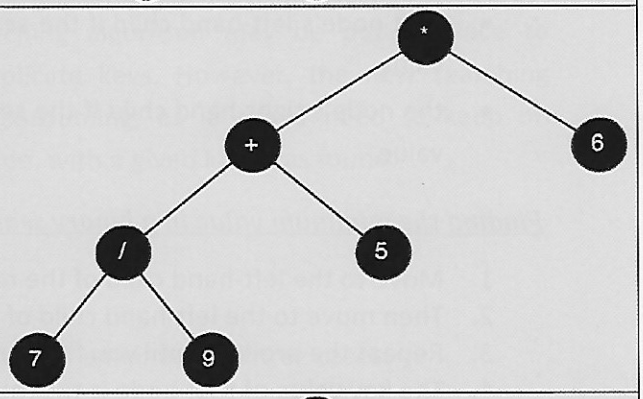
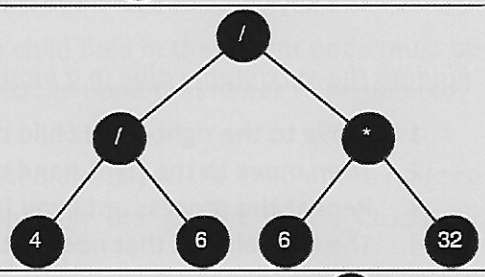
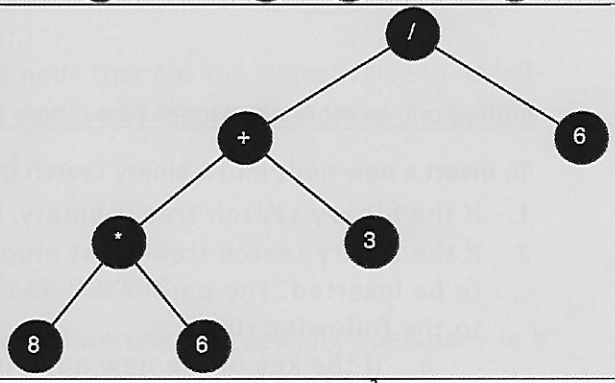
| Expression (infix). inorder traversal | Expression (prefix). postorder traversal | Expression (postfix). preorder traversal | Tree |
|---|---|---|---|
| (2+3)*6 | * + 2 3 6 | 2 3 + 6 * |  |
| ((7/9)+5)*6 | * + / 7 9 5 6 | 7 9 / 5 + 6 * |  |
| ((4/6)/(6*32) | / / 4 6 * 6 32 | 4 6 / 6 32 * / |  |
| ((3*6)+3)/6 | / + * 8 6 3 6 | 8 6 * 3 + 6 / |  |

**Table 5.8: Various expressions their notations and the equivalent binary trees[3]**

[3] Tool used: http://cstar.iiit.ac.in/~kkishore/DSVL/exp6/exptree.swf

### 5.1.17 Sketch binary trees

| Exit skills. Students should be able to[1]: |
| --- |
| Sketch diagrams showing the resulting binary tree after adding a new data item, adding one or more new nodes or removing one or more nodes. |

*Adding a new data item in a binary search tree.*

Adding a new data item means to add a new object. An object is added in the correct position of the tree as a node according to its key value. Thus, adding a new data item is similar to adding a new node.

*Searching for a particular data item in a binary search tree*

**Searching** for a particular data item involves comparing the value to be found with the key value of a node, and following:

- that node's left-hand child if the search value is smaller than the current node's key value.
- the node's right-hand child if the search value is greater than the current node's key value.

*Finding the minimum value in a binary search tree*

1. Move to the left-hand child of the root.
2. Then move to the left-hand child of that child.
3. Repeat the process until you find a node that has no left-hand child.
4. The key value of that node is the minimum value in the binary search tree.

*Finding the maximum value in a binary search tree*

1. Move to the right-hand child of the root.
2. Then move to the right-hand child of that child.
3. Repeat the process until you find a node that has no right-hand child.
4. The key value of that node is the maximum value in the binary search tree.

*Adding one or more new nodes to a binary search tree.*

To **insert** a new node into a binary search tree, follow the steps:

1. If the binary search tree is empty, insert the new node at the root.
2. If the binary search tree is not empty, follow the root to the parent of the node to be inserted. The parent will be a leaf node. Insert the new node, according to the following rules:
    a. If the key of the new node is smaller than the key of the parent node, then connect the new node as the parent's left-hand child.
    b. If the key of the new node is greater than the key of the parent node, then connect the new node as the parent's right-hand child.

To follow the root, simply compare the key value of the node that needs to be inserted with the root's node key value. If it is smaller, then move to the left-hand subtree. If it is greater, then move to the right-hand subtree. Continue with the next node and repeat the above process, until a leaf node is reached.

*Duplicate keys*

To deal with duplicate keys two methodologies may be applied:

1. The first and safest is to forbid the existence of duplicate keys. The candidate numbers of IB students, the tax identification numbers, the car license and registration numbers etc. are all unique and are used as key fields for searching, deleting and adding operations.
2. The second is to modify the insertion process so as to insert a node with a duplicate key as the right-hand child of the node with the same key. This will cause minor problems during the searching process, since when the first node, with a given key, is found the searching algorithm will stop and return the requested data. Of course this problem is solvable. A new searching algorithm may be put in place to accommodate for the existence of duplicate keys. However, the new searching algorithm would be a bit more time-consuming, as it would need to keep on searching the tree even after the first node, with a given key, was found.

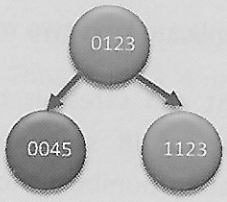
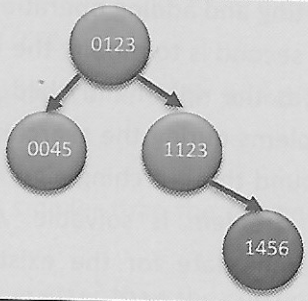
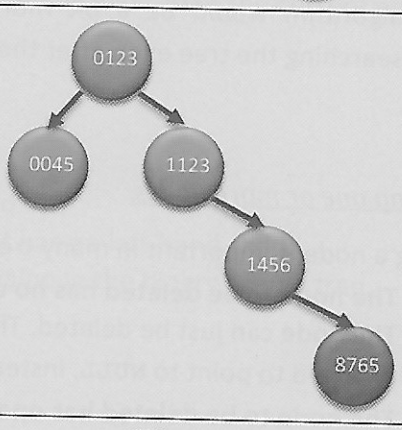
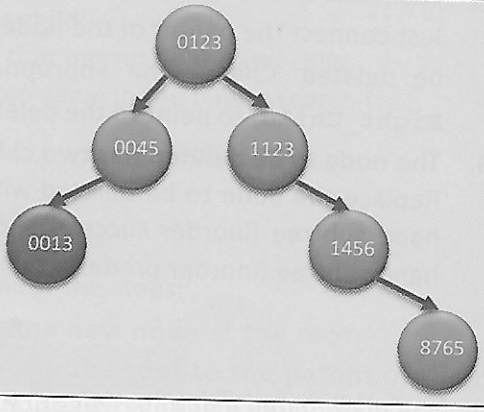
*Removing one or more nodes.*

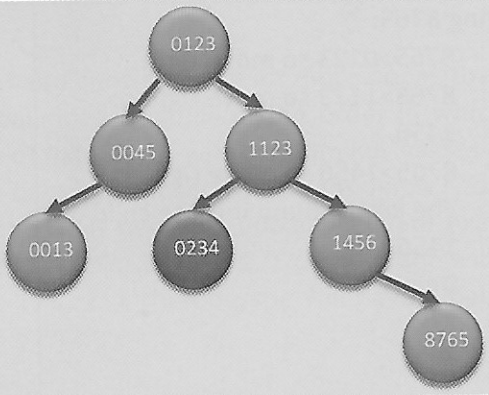
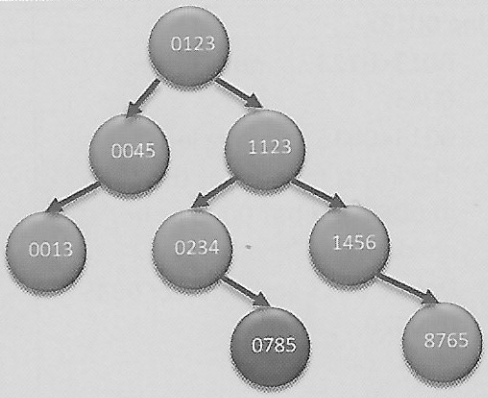
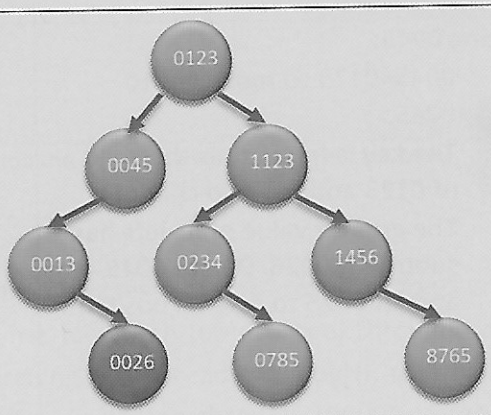
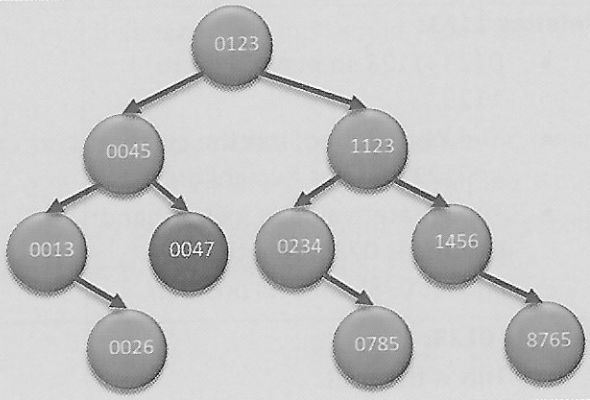
Deleting a node is important in many tree applications and involves three cases:

1. **The node to be deleted has no children:**
   The node can just be deleted. The appropriate child field in the parent node must be changed to point to `NULL`, instead of pointing to the node that needs to be deleted.
2. **The node to be deleted has one child:**
   Just connect the parent of the node to be deleted directly to the child of the node to be deleted. Change the appropriate reference in the parent (`Left_Child` or `Right_Child`) to point to the deleted node's child.
3. **The node to be deleted has two children:**
   Replace the node to be deleted with the node that has the largest value in its left-hand subtree (inorder successor) or the node with the smallest value in its right-hand subtree (inorder predecessor).

*Example of adding and deleting nodes in a binary search-tree*

The following table provides a detailed example of inserting and deleting operations in a binary search tree:

| | |
|---|---|
| **Inserting 0123:**<br>• The tree is empty so 0123 is the root node. |  |
| **Inserting 1123:**<br>• 1123>0123 so 1123 goes to the right. |  |
| **Inserting 0045:**<br>• 0045<0123 so 0045 goes to the left. |  |
| **Inserting 1456:**<br>• 1456>0123 so move right to 1123.<br>• 1456>1123 so 1456 goes to the right. |  |
| **Inserting 8765:**<br>• 8765>0123 so move right to 1123.<br>• 18765>1123 so move right to 1456.<br>• 8765>1456 so 8765 goes to the right. |  |
| **Inserting 0013:**<br>• 0013<0123 so move left to 0045.<br>• 0013<0045 so 0013 goes to the left. |  |

| | |
|---|---|
| **Inserting 0234:**<br>• 0234>0123 so move right to 1123.<br>• 0234<1123 so 0234 goes to the right. |  |
| **Inserting 0785:**<br>• 0785>0123 so move right to 1123.<br>• 0785<1123 so move left to 0234.<br>• 0234<0785 so 0785 goes to the right. |  |
| **Inserting 0026:**<br>• 0026<0123 so move left to 0045.<br>• 0026<0045 so move left to 0013.<br>• 0026>0013 so 0026 goes to the right. |  |
| **Inserting 0047:**<br>• 0047<0123 so move left to 0045.<br>• 0047>0045 so 0047 goes to the right. |  |

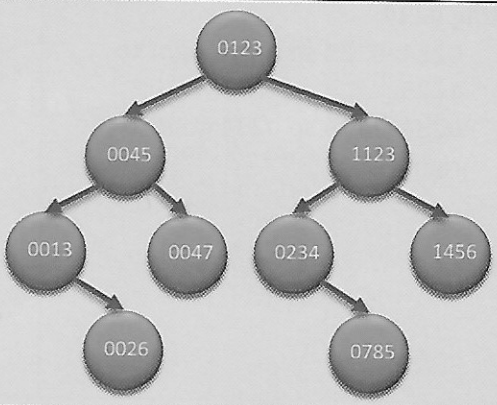
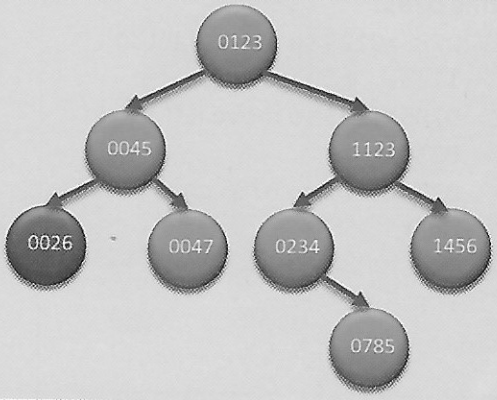
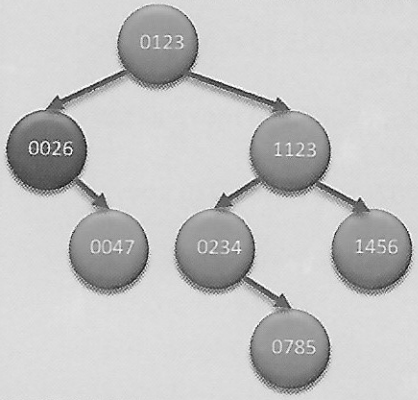
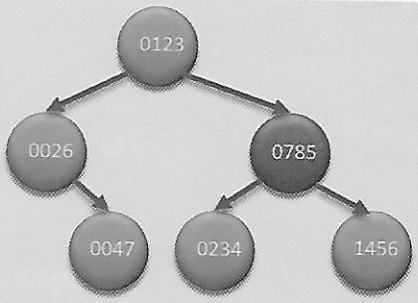
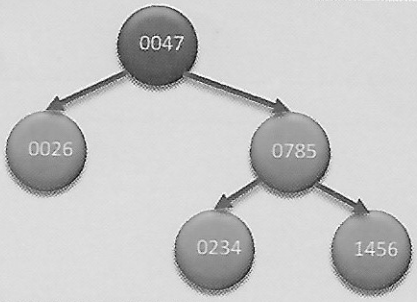
| | |
|---|---|
| **Deleting 8765:**<br>• 8765>0123 so move right.<br>• 8765>1123 so move right to 1456.<br>• 8765>1456 so move right.<br>• The key is found it is the child of 1456.<br>• Delete 8765. |  |
| **Deleting 0013:**<br>• 0013<0123 so move left to 0045.<br>• 0013<0045 so move left.<br>• The key is found; it is the child of 0045 and has 0026 as its child.<br>• Delete 0013 and connect 0045 to 0026. |  |
| **Deleting 0045:**<br>• 0045<0123 so move left to 0045.<br>• The key is found; it is the child of 0123 and has two children.<br>• The greater value of its left-hand subtree is 0026. Delete 0045 and let 0026 to take its position. |  |
| **Deleting 1123:**<br>• 0123<1123 so move right to 1123.<br>• The key is found; it is the child of 0123 and has two children.<br>• The greater value of its left-hand subtree is 0785. Delete 1123 and let 0785 take its position. |  |
| **Deleting 0123:**<br>• This is the root.<br>• Replace it with the greater value of its left-hand subtree.<br>• Delete 0123 and let 0047 to take its place. |  |

**Table 5.9: Adding and deleting node operations in a binary search tree**[1]

*Balanced trees*

An unbalanced tree is a tree whose left or right-hand subtree has a lot more nodes than the other subtree. Binary search trees become unbalanced because of the order in which the data items are inserted. In most cases, insertion of data items with random key values result in, more or less, balanced trees. However, if the data items inserted present an ascending or descending sequence of their key values, then the trees becomes unbalanced. For example, if one enters the data items with key values **1**, **2**, **3**, **4**, **5** or **5**, **4**, **3**, **2**, **1** then he/she will have an unbalanced binary tree. But if one enters **3**, **1**, **5**, **2**, **4** then he/she will have a balanced tee.

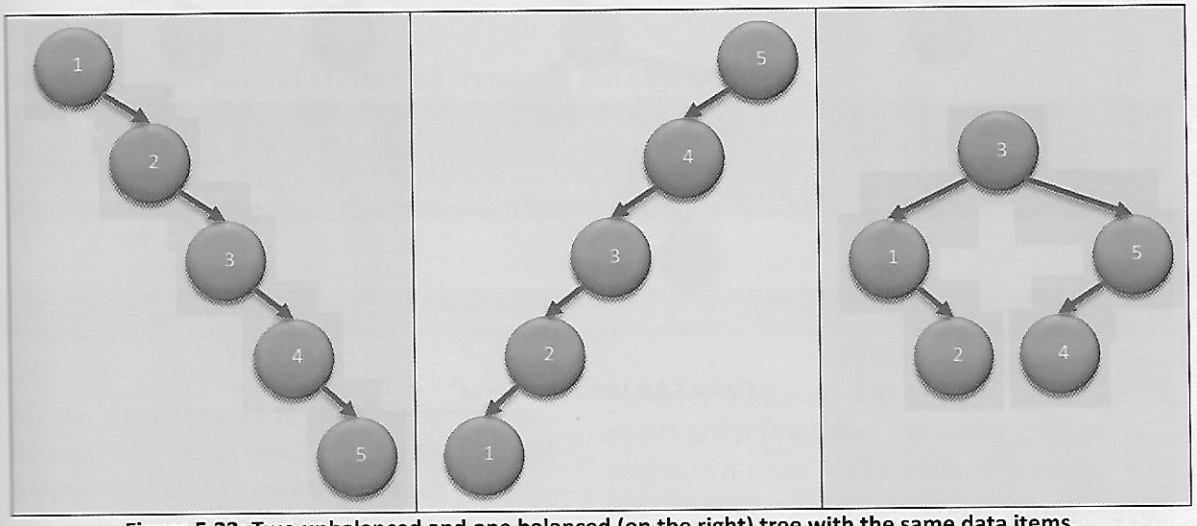



**Figure 5.22: Two unbalanced and one balanced (on the right) tree with the same data items.**

Inserting **1**, **2**, **3**, **4**, **5** or **5**, **4**, **3**, **2**, **1** are two extreme cases that both result to trees with no branches. These trees act like linked lists. One has to search (on average) through half the items to find the data item that he/she is searching. So instead of O(logN), of a balanced tree, one ends up with the O(N) of a linked list. Searching through 100000 items in an unbalanced tree requires 50000 comparisons. In a balanced tree this would require only 17 comparisons.

Suppose one wanted to insert "**ABCDEFGHIJKLMOPQRSTUVWXYZ**" in two different binary search trees, Figures 5.23 and 5.24. To find **J** in the first binary search tree he/she would need to follow the path **Y**, **G**, **S**, **H**, **P**, **O**, **L**, **I**, **K**, **J** (9 comparisons). To find **J** in the second tree he/she would need to follow the path **K**, **B**, **F**, **H**, **I**, **J** (5 comparisons).

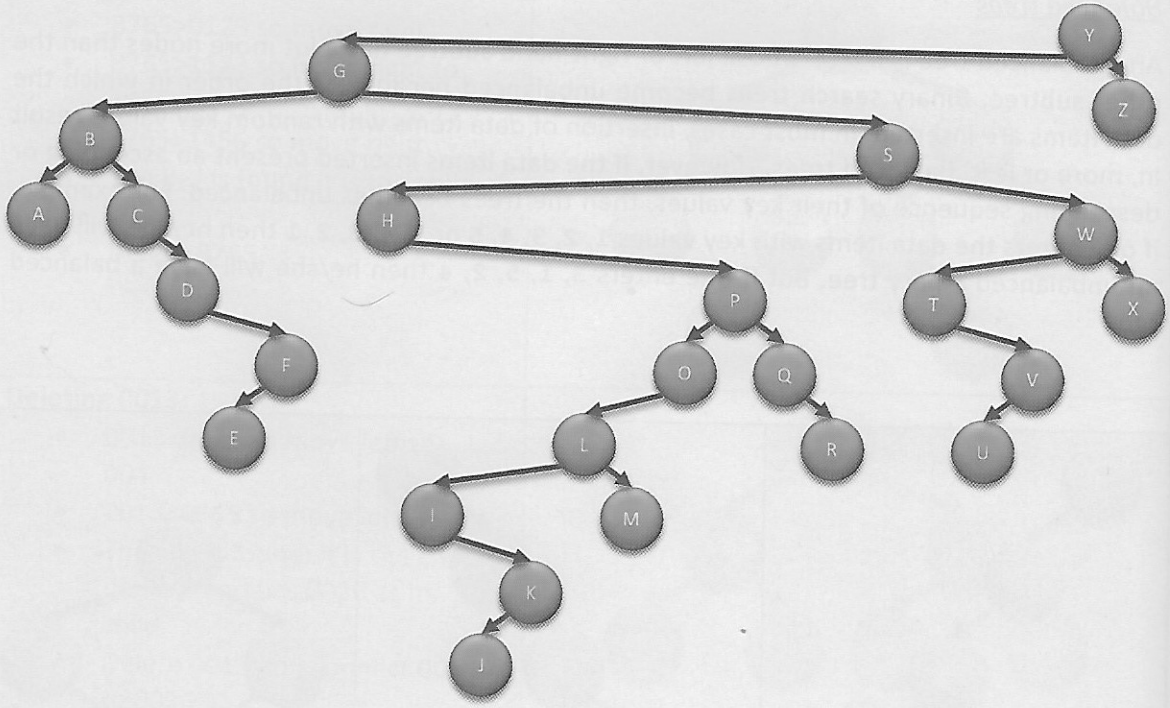



**Figure 5.23: Unbalanced tree**[4]

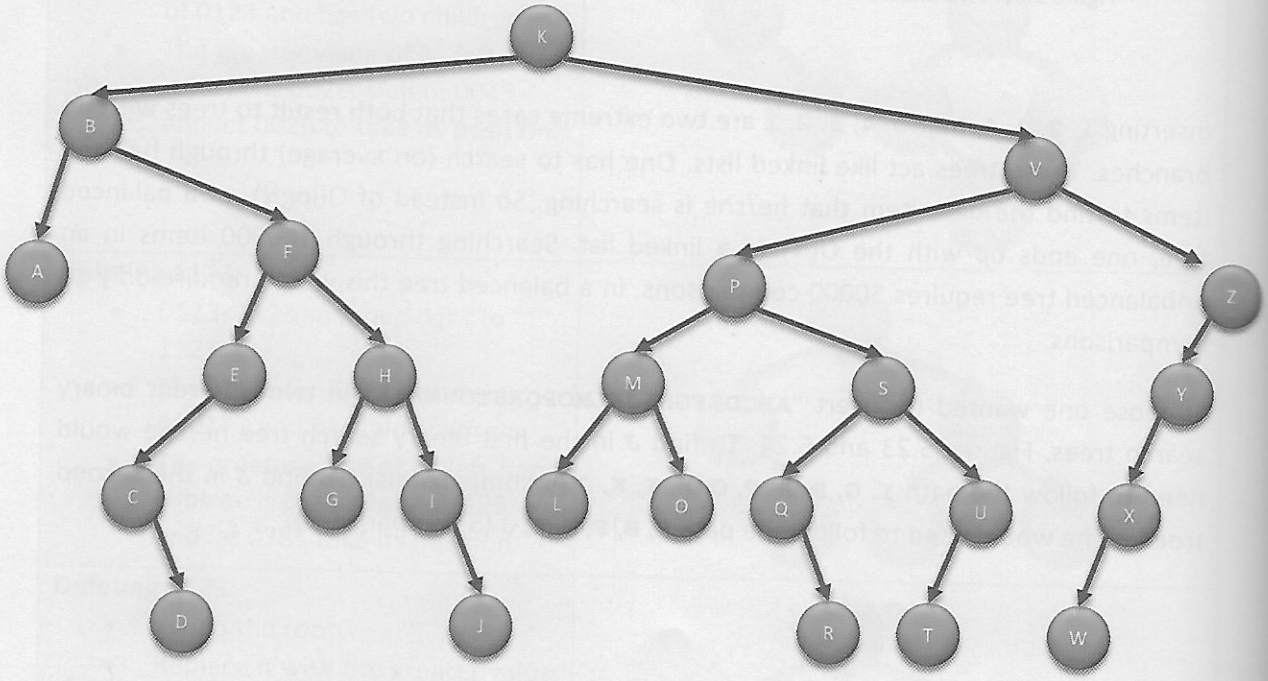



**Figure 5.24: Better balanced tree**

[4] Tool used: http://www.algomation.com/algorithm/binary-tree-insert-delete-display

### 5.1.18 Definition of the term dynamic data structure

| Exit skills. Students should be able to[1]: |
|---|
| Define the term dynamic data structure. |

A dynamic data structure changes its size at execution time as required by its elements. Allocation and de-allocation of memory is controlled by the data structure.

### 5.1.19 Comparison of static and dynamic data structures

| Exit skills. Students should be able to[1]: |
|---|
| Compare the use of static and dynamic data structures. |

| Dynamic and Static data structures | |
|---|---|
| **DYNAMIC** | **STATIC** |
| **Memory is allocated to the data structure dynamically i.e. at run-time. An example of a dynamic data structure is a stack implemented using linked lists.** | **Memory is allocated at compile time. The size is predefined and can never change during run-time. An example of a static data structure is a stack implemented using arrays.** |
| **Advantages** | **Advantages** |
| • Makes the most efficient use of RAM as it only uses as much memory as it needs.<br>• One does not need to know or decide upon the size of the data structure in advance. | • The memory allocation is fixed and as such there will occur no problems when adding or removing data items.<br>• Easier to program as there is no need to check upon the data structure size.<br>• The space reserved in RAM will always be available, in order to be used by the data structure. |
| **Disadvantages** | **Disadvantages** |
| • Given that the memory allocation is dynamic, it is likely that the structure will 'overflow' should it exceed its allowed limit or 'underflow' should it become empty.<br>• In most cases algorithms with dynamic data are slower, during execution, than algorithms with static data structures.<br>• Random access is not allowed and elements should be visited sequentially. | • Can be very inefficient as the memory for the data structure is predefined.<br>• Even when the array has no data elements in it, it still takes up the RAM space that was allocated at compile time.<br>• Sometimes it is difficult to predict the required array size.<br>• In a sorted array, inserting a new element in the correct position or deleting an existing one, requires |

| | |
|---|---|
| • As such, there is no way to implement binary search.<br>• More complicated to program as the software needs to keep track of its size and data item locations at all times. | shifting of other elements. |

**Table 5.10: Comparison of dynamic and static data structures**

| Data Structure | Time Complexity | | | | | | | | Space Complexity |
|---|---|---|---|---|---|---|---|---|---|
| | Average | | | | Worst | | | | Worst |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion | |
| Array | O(1) | O(n) | O(n) | O(n) | O(1) | O(n) | O(n) | O(n) | O(n) |
| Binary Search Tree | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |
| Double-Linked List | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Queue | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Singly-Linked List | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Stack | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) |

**Table 5.11: Efficiency of various data structures**

### 5.1.20 Suitable structures

**Exit skills. Students should be able to[1]:**

**Suggest a suitable structure for a given situation.**

An Abstract Data Type (ADT) is just an abstract conceptual tool. All data structures can be used to implement ADTs. A (static) linked list can be implemented using an array and an array-type structure can be implemented using a linked list. Building a linked list using an array is the option for primitive level languages and assembly languages that require fixed size data structures and do not support dynamic memory allocation. ADTs are conceptual models that abstract their fundamental data structures (the data structures that are used to implement them) and are used with some specific purpose in mind.

Stacks and queues are examples of ADTs that may be implemented either by using arrays or linked lists as their fundamental data structures. The important thing for a queue is to have an `enqueue()` and a `dequeue()` method. This can be achieved regardless of the use of an array, a linked list or a double-ended linked list. For a dynamic implementation of a stack the `push()` method would be implemented using the `Linked_List.insertFirst(data)` call, while the `pop()` method would be implemented using the `node = Linked_List.deleteFirst()` call. The user of the stack just uses the methods, without bothering with the details of the implementation (abstraction for the user), while the programmer, who has actually implemented the stack methods, knows the underlying programing mechanisms.

Binary trees can be represented using arrays. In the array approach, the nodes are stored in an array and are not linked by references. The index of the node in the array matches the

position in the tree. The node at index 0 is the root node, the node at index 1 is the root's left-hand child, and so on. This method is not very efficient. Empty nodes and deleted nodes leave holes in the array, occupying RAM, and when the deletion of a node involves moving subtrees, a lot of elements of the array must change position, resulting in a very time consuming operation.

| Data structure | BENEFITS | DRAWBACKS |
|---|---|---|
| **Array** | Fast insertion<br>Fast access of an element with a known index | Slow search<br>Slow deletion<br>Fixed size |
| **Ordered array** | More efficient search than unsorted array | Slow deletion<br>Slow insertion<br>Fixed size |
| **Stack** | Models physical stacks<br>Provides Last-In, First-Out (LIFO). | Slow access to other elements |
| **Queue** | Models physical queues<br>Provides First-In, First-Out (FIFO). | Slow access to other elements |
| **Linked list** | Fast insertion<br>Fast deletion | Slow search |
| **Binary search tree** | Fast search<br>Fast insertion<br>Fast deletion | |

## End of chapter example questions with answers

**Example 1**

Question

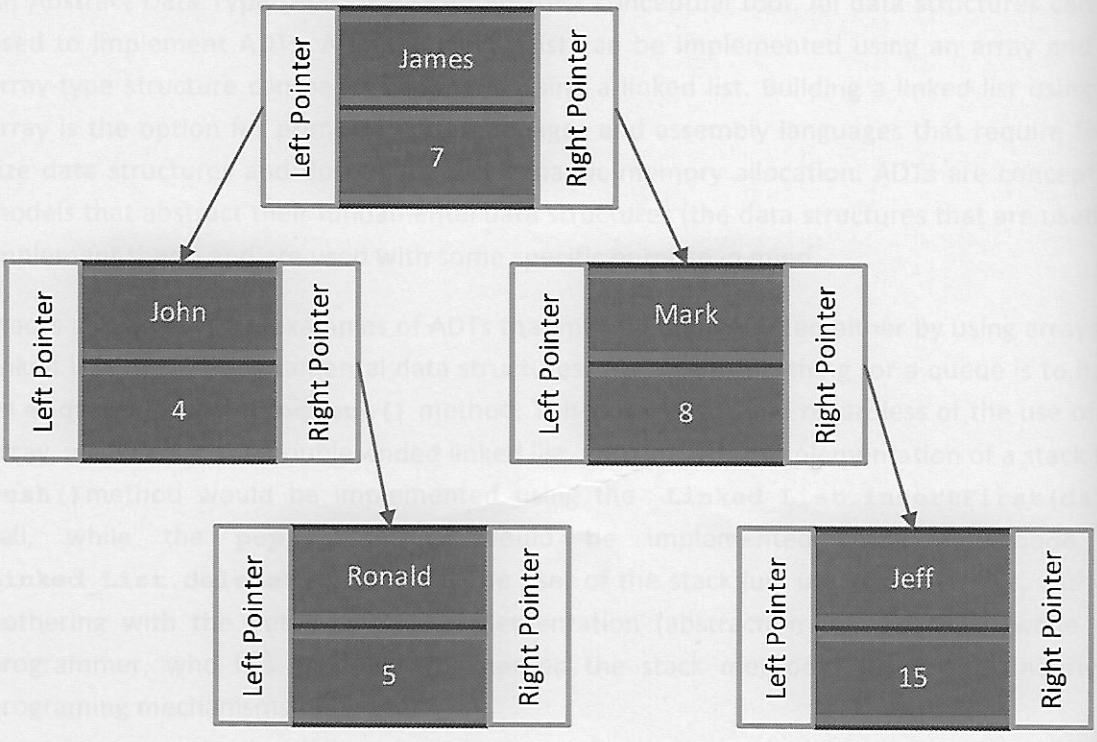
The following table shows basketball players and scores, from a game.

| NAME | SCORE |
|---|---|
| James | 7 |
| Mark | 8 |
| Jeff | 15 |
| John | 4 |
| Ronald | 5 |

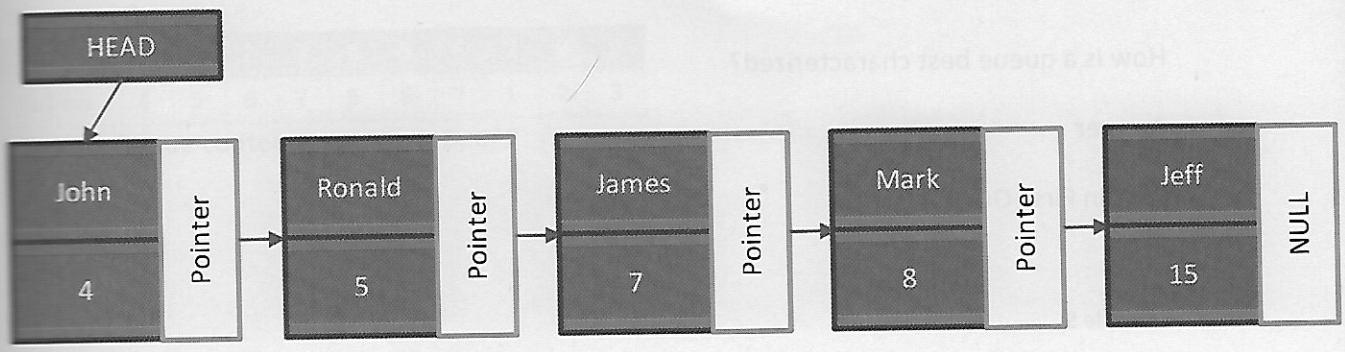
1. Construct a binary tree that will store the data of the table, given above, in the order of scores.
2. Construct the diagram of a linked list that will store the table data in ascending order.
3. Compare the use of the binary tree with a linked list.
4. State what will happen if we enter the scores data, in a binary search tree, in the following order: John, Ronald, James, Mark, Jeff.

Answer

1. The resulting binary search tree will be:



2. The resulting linked list will be:




3. Differences between the two data structures
    a. Binary search can be applied in binary search trees while linear search can be applied in linked lists. So it is faster to search a balanced binary search tree.
    b. A binary search tree uses two pointers for each node while a single linked list only one. So, a linked list needs less storage
4. The binary search tree will become unbalanced and the search for an element will be the same as in the linked list.

**Example 2**

Question

State three disadvantages of a recursive algorithm.

Answer

- It is more difficult to write.
- It is more difficult to maintain.
- An overflow error may occur if the stack runs out of space.

**Example 3**

Question

Explain how an element stored in a linked list could be found.

Answer

1. Start from the beginning (head) of the list.
2. Follow the pointer to the first node.
3. Compare the data in this node with the data to be found.
4. If the data is found, stop.
5. If the data is not found, follow the pointer of this node to the next node.
6. Repeat from step 3 until the data is found or the end of the list is reached.

**Example 4**

Question

How is a queue best characterized?

Answer

First-In First-Out

**Example 5**

Question

Given an empty queue `Queue`, what does it look like after the following instructions?

```
Queue.enqueue(6)
Queue.enqueue(8)
Queue.dequeue()
Queue.enqueue(3)
Queue.dequeue()
```

Answer

3

**Example 6**

Question

What is the reason for using a "circular queue"?

Answer

Reuse empty space.

**Example 7**

Question

Suppose there is a circular array-based queue implementation is capable of holding 10 elements. Show the array after the following code is executed:

```
loop m from 1 to 7
   enqueue(m)
end loop
loop m from 1 to 7
   enqueue(dequeue())
end loop
```

Answer

```
front = 7
rear = 3
```

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Value | 4 | 5 | 6 | 7 | 5 | 6 | 7 | 1 | 2 | 3 |

Circular Queue contents 1, 2, 3, 4, 5, 6, 7

**Example 8**

Question

Suppose there is a circular array-based queue implementation capable of holding 10 elements. Show the array after the following code is executed:

```
loop m from 1 to 3
   enqueue(m)
end loop
loop m from 1 to 2
   enqueue(dequeue())
end loop
loop m from 1 to 2
   enqueue(dequeue())
end loop
```

Answer

```
front = 4
rear = 6
```

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Value | 1 | 2 | 3 | 1 | 2 | 3 | 1 | | | |

Circular Queue contents 2, 3, 1

**Example 9**

Question

Is it possible to implement a queue using two stacks?

Answer

Yes. Two stacks are needed: an input and an output stack. All elements, at any time, must be either in the input or the output stack. When enqueuing, the elements are pushed in the input stack. When dequeuing, all the elements are popped from the input stack and pushed onto the output stack. The top element is then popped from the output stack to get the dequeued element. To add (enqueue) more elements one must pop the remaining elements from the output stack, push them to the input stack and add (push) the new element(s).

**Example 10**

Question

Which type of traversal always gives the sorted sequence of the elements in a binary search tree?

Answer

Inorder traversal

**Example 11**

Question

What is the maximum number of children that the largest element of a binary search tree must have?

Answer

1

**Example 12**

Question

What is the maximum number of children that the smallest element of a binary search tree must have?

Answer

1

**Example 13**[5]

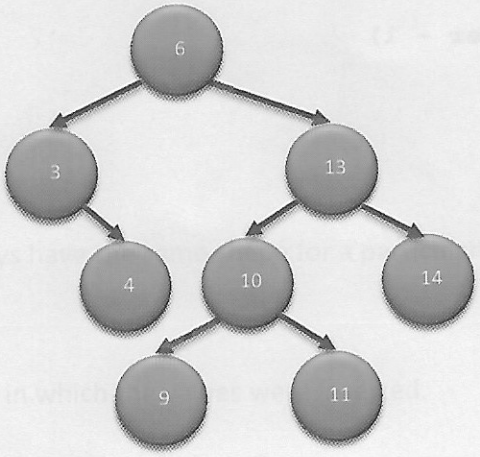
Question

Draw a binary search tree such that:

- each node stores a single number and
- a preorder traversal yields 6, 3, 4, 13, 10, 9, 11, 14 and
- a postorder traversal yields 4, 3, 9, 11, 10, 14, 13, 6.

[5] Tool used: http://btv.melezinek.cz/binary-search-tree.html

Answer



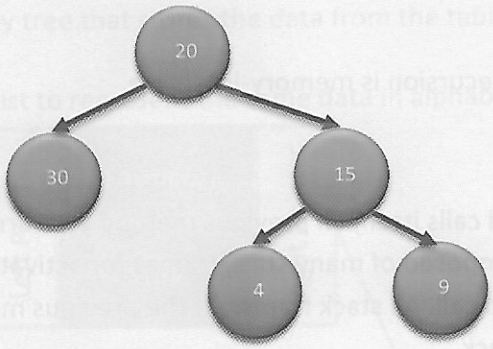

**Example 14**[6]

Question

Draw a binary tree such that:

- each node stores a single number and
- an inorder traversal yields **30, 20, 4, 15, 9** and
- a preorder traversal yields **20, 30, 15, 4, 9**.

Answer




**Example 15**

Question

What value does method **a** return when called with a value of **4**?

```
a(number)
```

[6] Tool used: http://btv.melezinek.cz/binary-search-tree.html

```
if (number <= 1) then
  return 1
else
  return a * a(number - 1)
end if
```

Answer

24 (i.e.: 4*3*2*1 = 24)

**Example 16**

Question

Which of the following data structures is *not* a dynamic data structure?

1. Array.
2. Binary tree.
3. Linked list.
4. Stack.

Answer

Array

**Example 17**

Question

Describe why the use of recursion is memory-intensive.

Answer

When a recursive method calls itself, all previous method calls are still open. The call stack (or execution stack) is composed of many stack frames (or activation records). Each stack frame relates to a method call. All stack frames of the previous method calls still occupy space in the execution stack.

**Example 18**

Question

Identify the type of linked list that:

1. starts with a pointer to the first node and
2. contains a pointer from each node to the next node and
3. in which the pointer in the last node points to the first node.

Answer

Circular, singly-linked list.

**Example 19**

Question

Do binary search trees always have the same shape for a particular set of data?

Answer

No, it depends on the order in which the values were inserted.

**Example 20**

Question

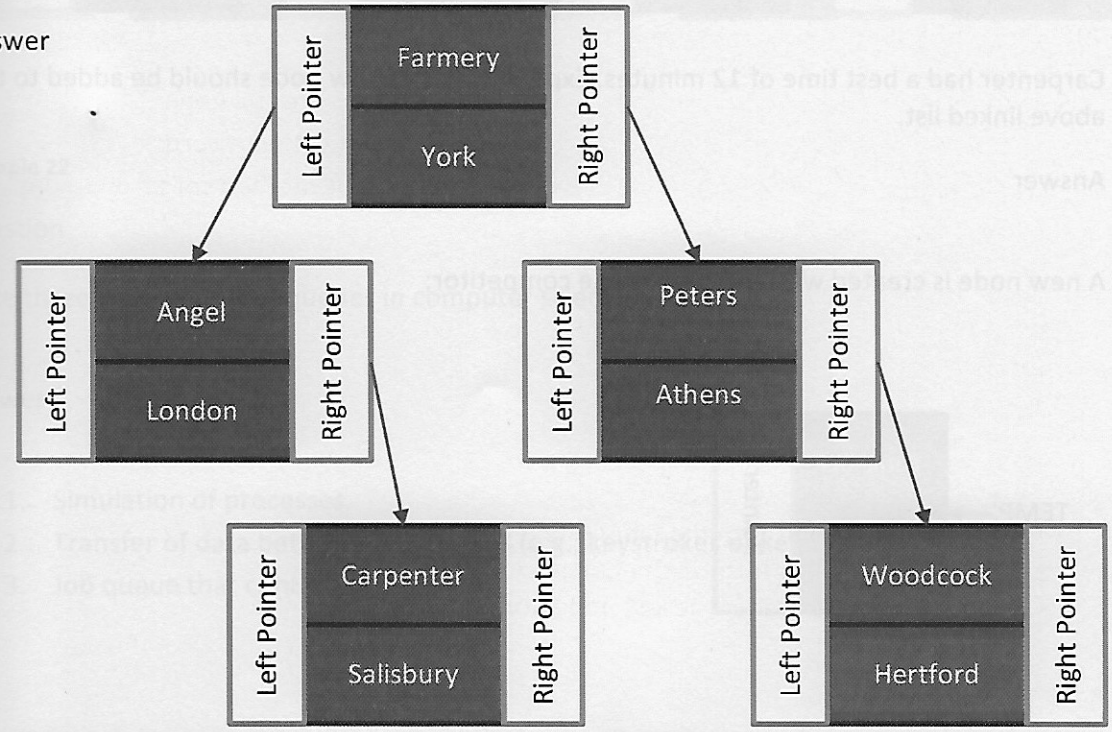
A piece of software finds the probable origin of a last name. When a user enters his/her last name he/she can find its origin. The following table shows some examples:

| Last name | Origin |
|---|---|
| Farmery | York |
| Peters | Athens |
| Woodcock | Hertford |
| Angel | London |
| Carpenter | Salisbury |

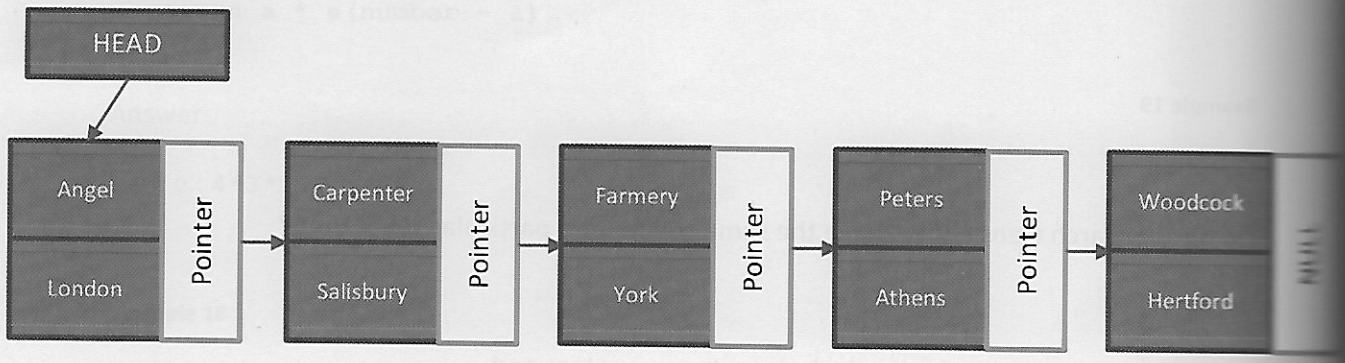
1. Construct the binary tree that stores the data from the table above in alphabetical order by last name.
2. Construct a linked list to represent the same data in alphabetical order by last name.

Answer

1.



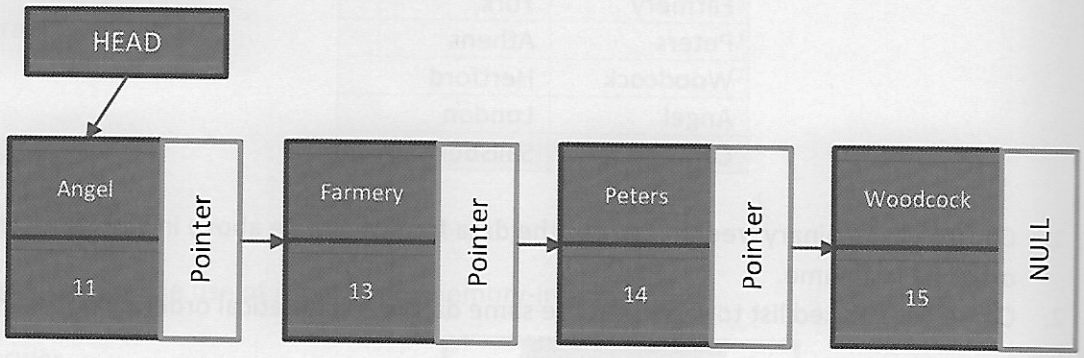
2.




**Example 21**

Question

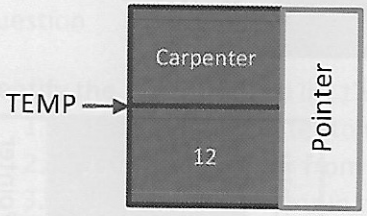
An application records the personal best times (in minutes) of competitors in a tournament. The program stores the data as they arrive in a sorted linked list according to the best times. Part of the data structure is given below:



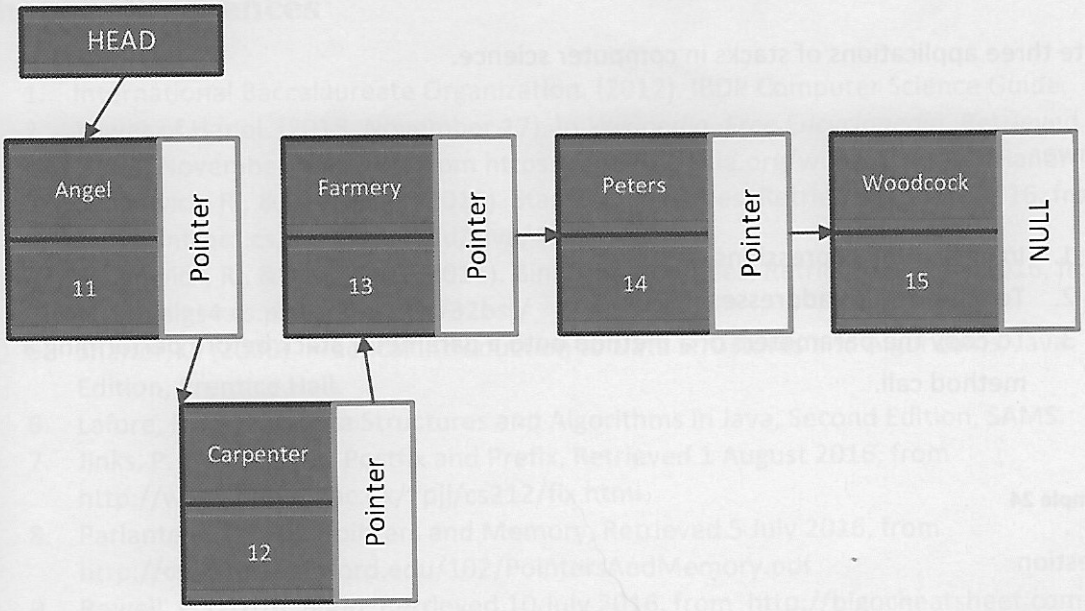

Carpenter had a best time of 12 minutes. Explain how this new node should be added to the above linked list.

Answer

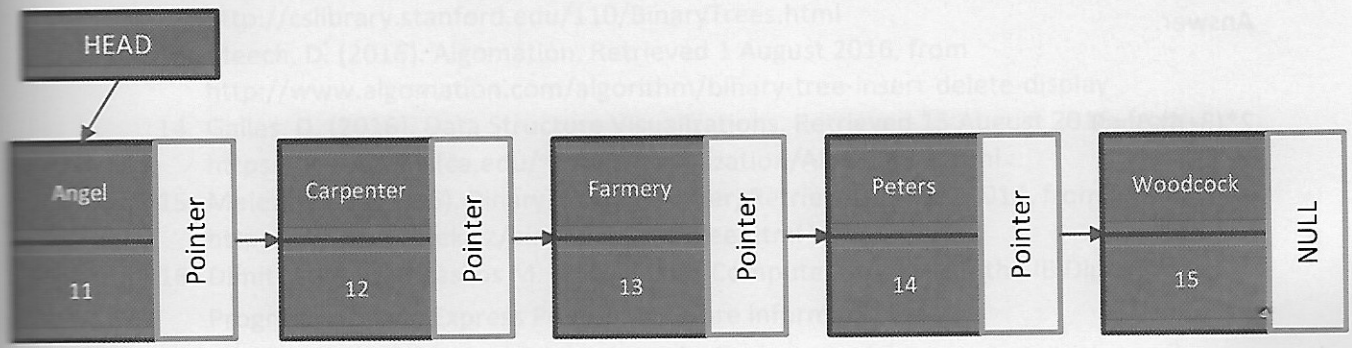
A new node is created with the data of the competitor:



Then, the pointers are adjusted:




The resulting linked list is the following:




**Example 22**

Question

State three applications of queues in computer science.

Answer

1. Simulation of processes.
2. Transfer of data between I/O devices (e.g.: keystrokes of keyboard).
3. Job queue that contains jobs to run.

**Example 23**

Question

State three applications of stacks in computer science.

Answer

1. In evaluating expressions.
2. To store return addresses.
3. To copy the parameters of a method onto a parameter stack before performing a method call.

**Example 24**

Question

Calculate the value of the following postfix expression:

2 3 4 + * 5 –

Answer

2*(3+4)-5 = 9

## Chapter References


1. International Baccalaureate Organization. (2012). IBDP Computer Science Guide.
2. Tower of Hanoi. (2015, November 17). In *Wikipedia, Free Encyclopedia*. Retrieved 14:03, November 17, 2014, from https://en.wikipedia.org/wiki/Tower_of_Hanoi
3. Sedgewick, R., & K. Wayne. (2016). Stacks and Queues, Retrieved 10 July 2016, from http://introcs.cs.princeton.edu/java/43stack/
4. Sedgewick, R., & K. Wayne. (2016). Binary search trees, Retrieved 10 July 2016, from http://algs4.cs.princeton.edu/32bst/
5. Shaffer, C. (2009). Practical Introduction to Data Structures and Algorithms, Java Edition, Prentice Hall.
6. Lafore, R. (2002). Data Structures and Algorithms in Java, Second Edition, SAMS.
7. Jinks, P. (2012). Infix, Postfix and Prefix, Retrieved 1 August 2016, from http://www.cs.man.ac.uk/~pjj/cs212/fix.html
8. Parlante N. (2000). Pointers and Memory, Retrieved 5 July 2016, from http://cslibrary.stanford.edu/102/PointersAndMemory.pdf
9. Rowell, E. (2016). Big-o, Retrieved 10 July 2016, from http://bigocheatsheet.com/
10. Kothapalli, K. (2016). Expression trees, Retrieved 15 July 2016, from http://cstar.iiit.ac.in/~kkishore/DSVL/exp6/exptree.swf
11. Parlante N. (2001). Linked List Basics, Retrieved 15 July 2016, from http://cslibrary.stanford.edu/103/LinkedListBasics.pdf
12. Parlante N. (2001). Binary trees, Retrieved 15 July 2016, from http://cslibrary.stanford.edu/110/BinaryTrees.html
13. Meech, D. (2016). Algomation, Retrieved 1 August 2016, from http://www.algomation.com/algorithm/binary-tree-insert-delete-display
14. Galles, D. (2016). Data Structure Visualizations, Retrieved 15 August 2016, from https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
15. Melezinek, J. (2016). Binary Tree Visualizer, Retrieved 15 July 2016, from http://btv.melezinek.cz/binary-search-tree.html
16. Dimitriou K. Hatzitaskos M. (2015). Core Computer Science for the IB Diploma Program. Athens: Express Publishing. More information at: https://www.expresspublishing.co.uk/gr/en/content/core-computer-science-ib-diploma-program
17. Halim, S., F. Halim, et al. (2016). Visualgo, Retrieved 1 August 2016, from http://visualgo.net/

# TOPIC 6 – Resource management

© IBO 2012

## Topic 6 — Resource management

## 6.1 Resource management

### System resources

### 6.1.1 Identification of critical resources

| Exit skills. Students should be able to[1]: |
| --- |
| Identify the resources that need to be managed within a computer system. |

System resources include computer hardware, software, trained personnel and supporting infrastructure. A critical factor when dealing with computer systems is the management of resources. Most resources present limited availability and should be managed with caution. In most cases the Operating System (OS) is responsible for the successful management of hardware and software resources. Different computer systems have dissimilar specifications, capabilities and purposes to fulfill. Some critical hardware resources include:

*Primary memory*

All processed data and instructions and all resulting data have to be stored in the primary memory. Primary memory is directly connected to the processor and feeds the processor with the required data by the fetch, decode, execute cycle data and instructions (Machine Instruction Cycle). Sometimes the primary memory is also referred as Immediate Access Store. Primary memory should be considered as an addressable matrix of cells with a unique address for each and every cell.

*RAM and Cache memory*

All data and instructions held in RAM may be altered at any time. There are two types of chips used for RAM: **static RAM** and **dynamic RAM**. Static RAM is a type of semiconductor memory that holds data for as long as there is power supply to the memory circuits. Data stored in a dynamic RAM semiconductor gradually leaks away and needs to be refreshed periodically. DDR-SDRAM (Double Data Rate - Synchronous Dynamic RAM) is a typical example of semiconductor technology used to build RAM chips in modern PCs. Static RAM is more expensive, needs more transistors per byte but is faster than DRAM. Both SRAM and DRAM are volatile and used concurrently nowadays. DRAM is used in large quantities in

[1] International Baccalaureate Organization. (2012). IBDP Computer Science Guide.

primary memory while SRAM is used in small quantities to speed up the overall performance by the **caching** technique, which balances the speed of DRAM with that of the much faster processor. The much faster SRAM is placed between the processor and the DRAM and directly feeds the processor. Data is moved from DRAM to SRAM and then to the processor and vice versa. This process has maximum benefits and performance when frequently-used instructions and data are stored in SRAM. The task of storing the correct data in SRAM is not always an easy task. Level 1 cache memory is usually built onto the processor while level 2 cache memory is on a separate chip located between the processor and the larger DRAM.

*ROM*

**Read Only Memory** (ROM) is non-volatile and slower than RAM. ROM is used to hold critical instructions used to start up a PC. A common use of ROM is to hold the Basic Input Output System (BIOS) which makes it possible for a PC to boot and sometimes to hold the entire OS for old small home computers (e.g.: Personal CP/M ROM-based version of CP/M 2.2 for small home computers developed by Digital Research Inc. and MSX Small ROM-based version of MS-DOS for Z80 home computers developed by Microsoft Corp.). ROM is also used in embedded microprocessors, microcontrollers and control systems.

*Secondary storage*

**Secondary storage, auxiliary storage and backing store** are terms that refer to hardware that provide data integrity, low cost, mass storage capacity and permanent storage. There are two broad types of secondary storage devices: the first uses **direct access** while the second uses **sequential access**.

*Direct access storage techniques*

The devices that fall into this category have the ability to access, retrieve and store any particular data without having to read through all previous data. Floppy disks, CD-ROMs, Hard disks, DVDs, USB sticks etc. belong to this category.

*Sequential access storage techniques*[2]

The devices that fall in this category have to read sequentially through all previous data before locating the requested data. Magnetic tapes, which belong to this category, are not only used in old science fiction movies, and are certainly not a dead technology. It is a major storage medium and recently IBM Research and Fuji Film have produced a magnetic tape with a record density of 123 billion bits of uncompressed data per square inch. This technology allows secure storage for 30 years, built-in data encryption, low cost and less energy consumption. Magnetic tapes are ideal for backup purposes.

*Processor speed*

A processor repeats the fetch, decode, execute cycle continuously as long as the computer is turned on. **MIPS** (Million Instructions Per Second) is used to measure the performance of a processor. MIPS is only an approximation of a processor's performance since it does not take into account the fact that sometimes a single instruction may operate on many operand

[2] IBM sets new tape storage record. (13, April, 2015). In *New Atlas*. Retrieved 19:05, June 13, 2016, from http://www.gizmag.com/ibm-tape-storage-record/36931/

fetches and stores, some instructions have a higher effectiveness than others, and some processors have the ability to execute several instructions simultaneously.

$$\textbf{MIPS} = \frac{Processor\ Clock\ Frequency\ (Master\ Clock\ MCLK)}{Average\ number\ of\ clock\ Cycles\ Per\ complete\ Instruction\ (CPI) * 1000000} = \frac{cycles/second}{\left(\frac{cycles}{instruction}\right) * 1000000} = \frac{million\ instructions}{second}$$

The **clock rate** refers to the frequency at which the processor is running and is commonly used as a rough indicator of the processor's performance. The unit used is **hertz** and in the case of multicore processors the clock rate is the same for all cores. The clock rate of modern processors is measured in **gigahertz** (GHz). It is very important to mention that the clock rate should not be used as a perfectly reliable measure of the performance of different processor families.

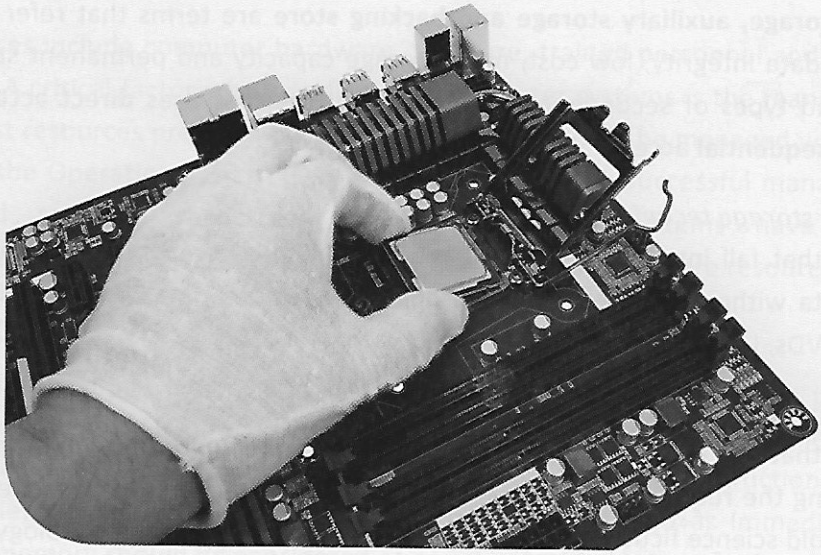


**Figure 6.1: Installation of a modern processor in a CPU socket on the motherboard**

*Bandwidth*

Memory bandwidth is the rate at which data can travel from SRAM and DRAM to the processor and vice versa and is essential to the performance of a CPU. It is expressed in millions of bits per second or in Mb per second (Mb/s). The **peak theoretical bandwidth**, which is typically one word per bus cycle, is not the same as the **sustained memory bandwidth**, which is less and is affected by various design features.

*Screen resolution*

All digital television sets, computer monitors, tablet touch-screens and mobile phone screens have a maximum number of distinct pixels that can be used to display video, pictures, text etc. This is mentioned as *Maximum resolution = Width $_x$ Height*, where Width is the number of distinct pixels in the horizontal dimension and Height is the number of distinct pixels in the vertical dimension. Most devices can support a number of different resolutions. A rule of thumb is that higher resolutions need more memory and more processing power.

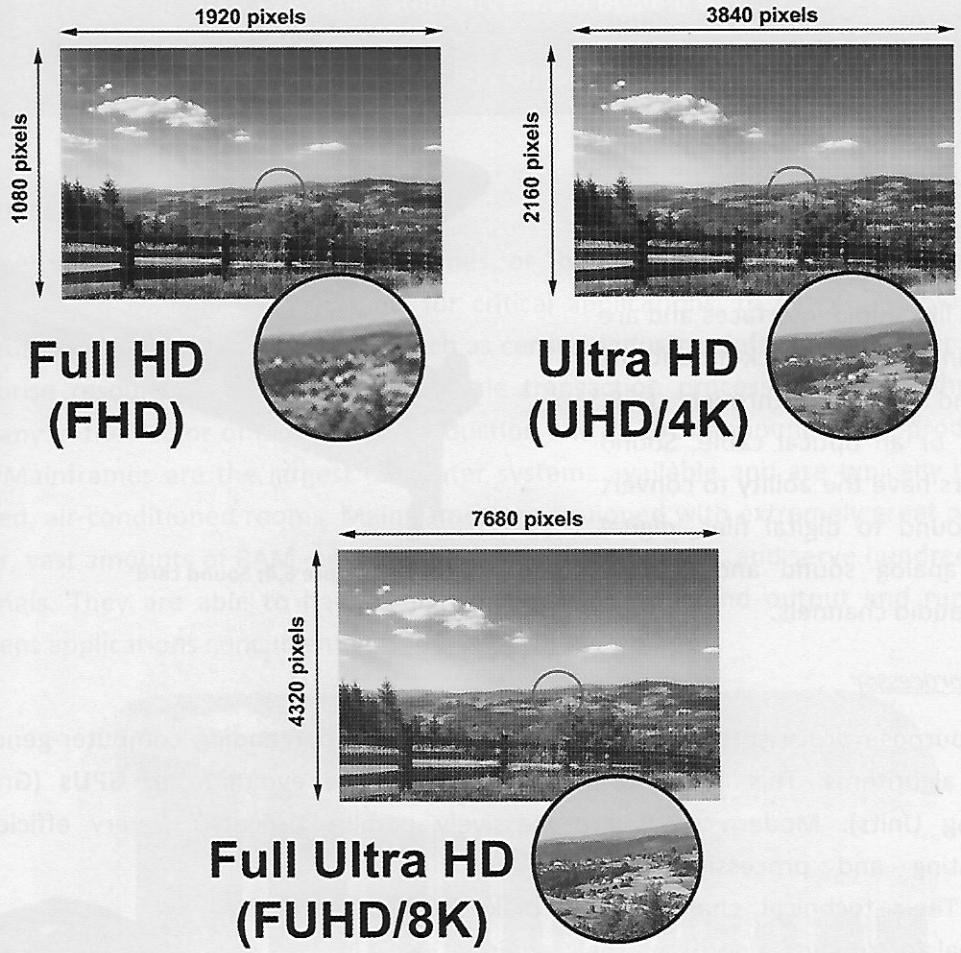



**Figure 6.2: TV display with resolution comparisons**

*Disk storage*

**Disk storage** is a general category of metal or plastic storage plates on which data can be recorded. The rotating disk(s) are mounted on a central spindle. Common disk storage devices are the hard disk drive (**HDD**), the floppy disk drive (**FDD**) and various optical disc drives. Nowadays HDD, solid state drives (**SSD**) and solid state hybrid drives (**SSHD**) are the main disk storage systems used in most computers. SSDs are very expensive, very fast, more

durable and consume less energy. HDDs are very cheap, present slower boot times, can cover all storage requirements and are cost-effective. SSHD share the benefits but also the disadvantages of HDDs and SSDs and they probably provide the best affordable combination of performance characteristics.

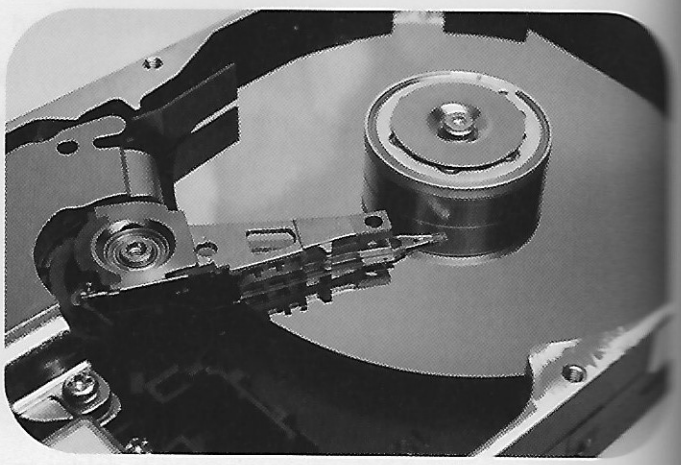


Figure 6.3: Hard disk

*Sound processor*

**Sound cards** facilitate the input, process and output of audio signals. In most cases they are integrated onto the motherboard, while some advanced models are sold as separate cards that use an expansion slot of the motherboard. Professional sound cards act like audio interfaces and are hosted in external rack-mountable units and connect through USB, FireWire, or an optical cable. Sound processors have the ability to convert analog sound to digital files, digital files to analog sound and process multiple audio channels.

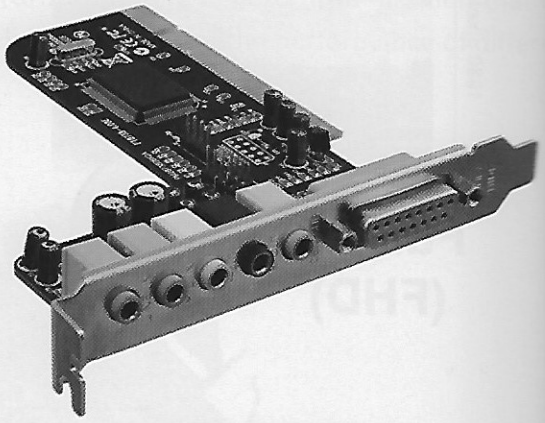


Figure 6.4: Sound card

*Graphics processor*

General-purpose processors are not efficient at running demanding computer-generated imagery algorithms. This is the primary reason for the evolution of **GPUs** (Graphics Processing Units). Modern GPUs are massively parallel processors, very efficient at manipulating and processing graphics and images. Their technical characteristics make them ideal for running algorithms that require processing of large blocks of graphics data and computer generated imagery. Their rapid evolution has been driven by the video game entertainment industry, which is a fast growing sector with great potential. Faster GPUs are sold as separate cards that use an expansion slot of the motherboard, while some low-end models are embedded on the motherboard or are integrated with the main processor circuit.

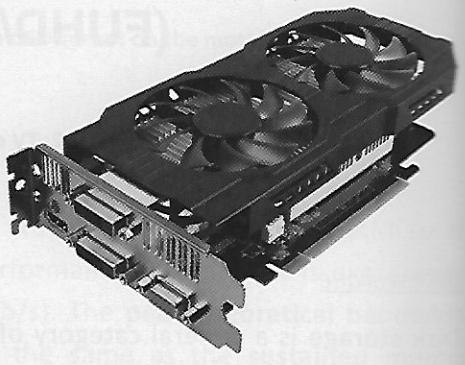


Figure 6.5: Graphics card

*Network connectivity*

Some computers have various network connectivity capabilities. A laptop which is equipped with both a wired **Network Interface Card** (NIC) and a wireless NIC will outperform a tablet which is only equipped with a wireless NIC. In this case, the laptop presents a better network connectivity solution. On the other hand, a lot of modern tablets are **4G** enabled meaning that they have **SIM** (Subscriber Identification Module) card slots. This greatly enhances their ability to work and stream various media on the move.

### 6.1.2 Availability of resources

| Exit skills. Students should be able to[1]: |
| --- |
| Evaluate the resources available in a variety of computer systems. |

*Mainframes*

Together with supercomputers, **mainframes**, or "big iron" are the flagships of computing. They are used by large organizations for critical applications, to handle large-bandwidth communication, bulk data processing such as census, industry, defense, consumer statistics, enterprise resource planning and large-scale transaction processing. IBM is the leading company in the sector of mainframe production and in 2015 announced the production of Z13[3]. Mainframes are the largest computer systems available and are typically housed in isolated, air-conditioned rooms. Mainframes are equipped with extremely great processing power, vast amounts of RAM, arrays of disks and backup tapes, and serve hundreds of user terminals. They are able to handle high volumes of input and output and run a lot of different applications concurrently.

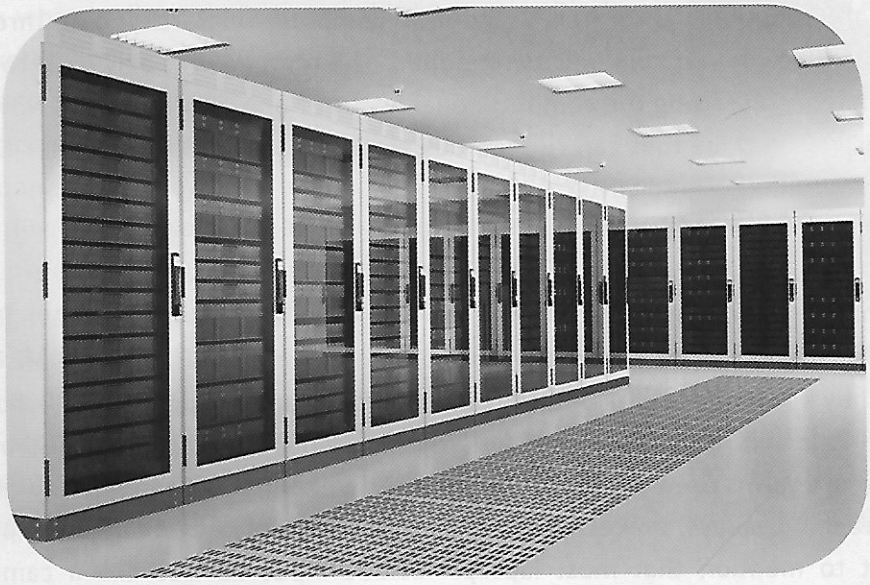


**Figure 6.6: Mainframe**

[3] The digital revolution demands a better server. In *IBM*. Retrieved 20:05, June 14, 2016, from http://www-03.ibm.com/systems/z/hardware/z13.html

*Supercomputers*

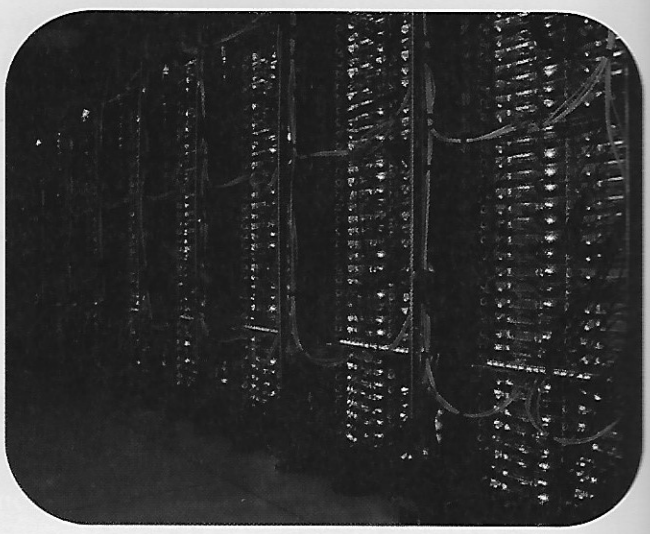


**Figure 6.7: A supercomputer**

They are very fast and expensive, and focus on mathematical calculations, weather forecasting, climate research, molecular modelling, scientific and engineering applications. The performance of a supercomputer is measured in floating-point operations per second (**FLOPS**). New **supercomputers** exceed 10 PFLOPS drawing on the power of more than 30000 processors. The first supercomputers were compact designs that used the local parallelism approach, while the latest designs are considered as massively parallel systems equipped with multiple arrays of computers and processors.

*Servers*

A **server** is software, hardware, or both and provides various services to clients. The client-server model is fundamental in computer networking and modern servers serve database, file, email, game, application etc. requests from various clients. Servers need multiple network connections for advanced performance, a lot of RAM to support multiple requests, fault tolerance and ease of repair without the need of shut down, advanced backup facilities, superior security characteristics and various automation capabilities.

*PCs*

Microcomputers and home computers were the first terms used to describe what we now call Personal Computers (**PC**). Years ago the computer hierarchy had three classes: mainframes, minicomputers and microcomputers. The term **PC** is widely used to describe a device capable of supporting the computational needs of one user at a time. Nowadays PCs are inexpensive solutions that can run various software applications. Popular OSs for PCs are MS Windows, MacOS and LINUX.

*Laptops*

The rapid change in working patterns and the increased need for mobility have favored the popularity of mobile devices. Modern laptops are not a compromise in comparison to a PC. Moreover, they have long-lasting batteries that can be used to support hours of computing. **Laptops**, mobile workstations, desktop replacements, sublaptops, notebooks and subnotebook are terms used to describe mobile PCs with different weights and dimensions, as well as graphic-displays, processors, RAM, secondary storage and battery capabilities. It is important to mention that most laptops include input devices like a camera and a microphone and output devices like speakers. Touch-screens are now widely used as input devices and facilitate tasks such as drawing and editing.

*Tablets*

A **tablet** is a mobile computer equipped with a touch-screen display which is used as both an input and an output device. Tablet users can take advantage of the touchscreen to enter text using the virtual keyboard or the built-in handwriting-recognition facility. The use of various gestures completely replaces the need for a mouse, while finger and stylus pens are widely used in all applications. iOS, Android and Windows are the major OS for tablets. Hybrid tablets have a detachable keyboard and closely resemble laptops. Tablets are equipped with sensors like fingerprint, three-axis gyro, GPS and accelerometers, front and rear cameras, Bluetooth for connecting peripherals, Wi-Fi for networking and powerful batteries. Specifications for high-end tablets include 2732x2048 pixels display, 12.9" screen, 128 GB internal storage, 3 GB RAM, slot for 16 to 128 GB microSD (Secure Digital) memory card, octa-core processor, 12.0 MP camera and 24-bit/192kHz sound. Various developers can develop apps for tablets and distribute them online through application stores like Apple's App Store and Google's Android Play/Store.

*PDAs*

**Personal digital assistants (PDA)**, were used until 2010 as electronic agendas, calendars and personal information systems. Nowadays smartphones offer all the capabilities once provided by PDAs. Common OS for PDAS were Palm OS, BlackBerry OS and Windows CE.

*Cell phones*

A **smartphone** or **smart phone** is a mobile phone (portable telephone) whose hardware components and software are managed by a mobile operating system. Nowadays smartphones combine features of PCs, PDAs, cameras, media players and GPS navigation units. All smartphones can access the Internet, are equipped with touchscreens that enable the user to interact directly with what is displayed, can run third-party applications and have photographic capabilities that approach those of mid-level point-and-shoot cameras. They also support high-speed mobile broadband through 4G LTE, Wi-Fi connection and Bluetooth connection. They also feature RFID solutions, motion sensors, a fingerprint sensor, accelerometer, gyro sensor, proximity sensor, compass, barometer and heart rate monitor, as well as mobile payment transactions and geotagging mechanisms. Modern smartphones use iOS, Windows and Android OSs. Various developers can develop apps for smartphones and distribute them online through application stores like Apple's App Store and Google's Android Play/Store. Specifications for high-end smartphones include 1440x2560 pixel display, 64 GB internal storage, 4 GB RAM, slot for 16 to 128 GB microSD (Secure Digital) memory card, octa-core processor and 24-bit/192kHz audio.

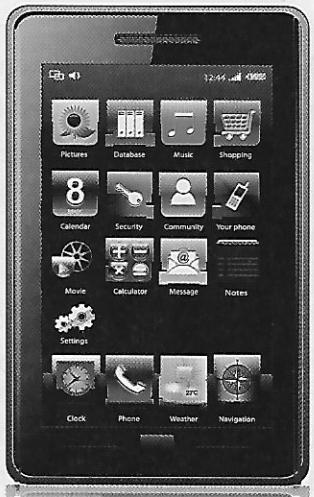



**Figure 6.8: A smartphone**

*Digital cameras*

A digital camera encodes images and videos digitally and stores them in the attached memory card for later reproduction. High-end products have a 30.4 Megapixels sensor and are equipped with an LCD viewfinder, as well as a GPS sensor. They support various image formats, such as JPEG and RAW, as well as video formats, such as MP4. They can record 4K video and connect to a computer system through a USB interface or Wi-Fi connection. They may also include HDMI output, analog stereo audio output and analog video output (NTSC/PAL). All the signal processing and control functions of a digital camera are performed by a dedicated specialized processor.




**Figure 6.9: A digital camera**

### 6.1.3 Limitation of resources

| Exit skills. Students should be able to[1]: |
| --- |
| Identify the limitations of a range of resources in a specified computer system. |

Sometimes when a computer system is active the request for services exceeds the availability of resources. Memory is a typical example of system resources. Every time one opens an application the OS is reserving a particular amount of memory that the program needs to operate. If a PC is equipped with 8 GB of RAM, the available memory to run various applications is around 6 GB because the OS and various programs that load during the startup process utilize a total of 2 GB. Although the OS will do its best to satisfy ones requests, the launching of more and more applications will lead to an "Out of Memory" message or a very slow computer.

In **computer-generated imagery (CGI)**, **rendering** is the process of generating a 2D or 3D graphic and exporting it to an image file. It also refers to the process of adding effects during the video editing process, in order to generate a final product. The rendering process involves complex mathematics and the solution of the rendering equation. This integral equation must be solved by rendering software to produce realistic graphics. Although the advances in personal computer resources allow an image to take less time to render than before, state-of-the-art image quality needs specialized solutions, such as **render farms**. These high performance computer systems are used by professionals to render computer generated imagery.

A **single core single processor** system may not be able to perform demanding tasks or run complex mathematical models efficiently. A multicore or a multiprocessor system may be more efficient. A single processor system may have an **IC (Integrated Circuit)** with multiple cores. In a **multiprocessor system** two or more ICs are mounted on the motherboard. Each

IC could have more than one core. A multiprocessor system is more expensive, needs complex configuration and usually runs multiple programs faster, whereas a **multicore system** usually runs a single program faster and is considered as the best choice for everyday users. This, oversimplified, statement assumes that the comparison takes place between ICs with the same characteristics and clock speeds. It is important to keep in mind that in a multicore system, the main cache memory is shared by all the cores.

A computer system may have one or more standalone **GPU**s. A single GPU is what most users opt for. It is an economical and powerful option that can serve as many as 4 different monitors, play HD video, and provide gaming capabilities. If someone wants to play an action game in a 4K resolution then using a single GPU may not allow the game to run smoothly. However, multiple GPUs can generate disturbing noise and high temperatures, especially under load. In most cases, two or three GPUs of the same type do not result in a double or triple increase in the performance respectively.

### 6.1.4 Problems with insufficient resources

| Exit skills. Students should be able to[1]: |
| --- |
| Describe the possible problems resulting from the limitations in the resources of a computer system. |

Thirty years ago, few computers could process more than one program at a time. This **single program operation** was the most common and the loading and running of each program was supervised by a simple OS like MS-DOS. Computer system resources were scarce and it took a long time to complete tasks that needed more than a single program to run in order to be completed.

As computer systems evolved, a slightly more sophisticated and complex type of computer operation appeared called **batch processing**. In batch processing some programs are batched together and then executed as a group, without the need of any intervention from the user. Only one program is actually running at a time, while the others are waiting for their turn. When it completes, the next program in the queue runs, and so on, until all the programs in the batch are run.

As computer systems evolved even more, and more than one programs could be loaded in main memory at the same time, ready to execute, a new type of operation emerged, called **multiprogramming**. In multiprogramming two or more programs may be loaded in the main memory. However, only one program will actually be executed by the CPU at any one point in time. All the other programs will be waiting for their turn. The idea behind multiprogramming is to maximize the use of CPU time (i.e. CPU idle time should be minimized). For example, if the currently running program is performing input/output tasks (eg. waiting for input from the user or drawing an image on the screen) the CPU will be idle, since it will not be needed. A multiprogramming OS will give control to one of the other programs residing in the main memory and needs to execute, thus reducing CPU idle time.

Even with multiprogramming, if there a number of programs loaded in the main memory and all are CPU intensive, without any idle time for I/O operations, the last program will have to wait for all the other programs to finish, before executing. Another problem that needs to be dealt with when there are multiple programs residing in main memory is memory fragmentation as programs are moved from and to the memory. Unix, developed by Bell Laboratories of AT&T, was one of the first OSs that supported multiprogramming. In a multiprogramming time-sharing environment, a lot of different users share a computer system simultaneously. This situation can cause numerous security problems, such as stealing/copying another's user programs/data or using system resources without proper accounting. In any case, the degree of security in a multiprogramming system is less compared to the security in a single-user dedicated system.

Further system development and evolutions introduced a new type of operation called **multitasking**. Multitasking is similar to multiprogramming with the subtle difference that tasks or processes (instead of whole programs) are performed simultaneously and share a common resource, for example one CPU. Each task finishes, before another takes up the CPU, as was the case in multiprogramming. However, tasks are a lot "smaller" than programs and as such are completed very quickly. Both multiprogramming and multitasking operating systems are **Central Processing Unit time sharing** systems. In older systems multiprogramming allowed one program as a whole to run until it was completed, while in newer systems multitasking best manages the utilization of CPU resources with the use of program fragments called processes. Since tasks are completed in a timely fashion, the illusion of parallelism is achieved. That is, all programs running on a computer appear to be operating at the same time to the user. What is happening however, is that small tasks, from each program, are completed very quickly by the CPU, making the programs appear as if they are all operating at the same time.

**Multiprocessing** refers to the hardware and means that a computer system has more than one CPU cores. This might mean multiple CPU dies or even multiple cores in one or more CPU dies. Since multiprocessing refers to hardware, whereas multiprogramming and multitasking refer to software, a system can be both multiprocessing, as well as multiprogramming or multitasking.

**Multithreading** is the ability of a program or an operating system to execute different parts of a program, called **threads**, simultaneously. The program has to be designed by the programmer in such a way that all the threads can be executed at the same time without interfering with each other. It is important to mention that several threads of a single process can share the CPU, in a single CPU system, or run in parallel in a multiprocessing or multicore system. Multithreading is widely used in applications that encompass a GUI. For example, in such an application, if a task requested by the user needed a long time to complete (because it was a complex mathematical computation or a network call, etc.), the GUI would be unresponsive ("freeze") until the task was completed. Multithreading, on the other hand, would allow the GUI of the application to be responsive and the user to have a better experience.

Another advanced mode of operation is **multi-access,** where a lot of users can interact with a one computer system through their terminals. The computer system may execute a number of programs and the connected users can interact with these programs. This mode of operation, of a computer system, allows the simultaneous connection of a number of terminals. Such a system must embrace the following functional characteristics: (a) multiline communication capabilities that will support simultaneous dialogues with the remote terminals; (b) concurrent execution of various programs with the ability to instantly switch from executing the program of one client to executing that of another; (c) ability to quickly find and make data stored on the hard disks available; (d) ability to protect all data from unauthorized access. Unix OS supports multi-access and provides the above mentioned characteristics.



### 6.1.5 Role of the Operating System (OS)

| Exit skills. Students should be able to[1]: |
| --- |
| Explain the role of the operating system in terms of managing memory, peripherals and hardware interfaces. |

**Operating Systems (OSs)** can be classified as single user, multi-user, multiprocessing, multitasking and multi-threading. Most OS are now written in C or C++ while older OS were written in low level languages. A few critical operations that are very important for the performance of an OS are still written in assembly. All OSs are collections of software and belong to the system software. An OS is a system, meaning that the collection of software that form an OS collaborate towards a common goal. Any computer system is a collection of hardware and software resources which provide input, process and output services. The OS is the core software that coordinates all these resources, on top of which all other software applications reside. The OS controls the execution of all other software. For example, it will locate where an application is saved on the HD, calculate the amount of RAM needed for the application to run, allocate the correct amount of RAM for the application, copy the application to the allocated RAM, etc.

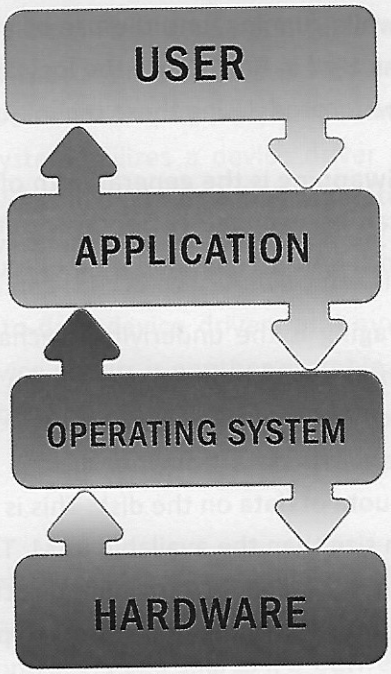



**Figure 6.10: Different levels of a computer**

Usually, a computer's RAM is small compared to its secondary storage. Memory management is the act of managing computer memory and every instruction that is executed in the arithmetic and logic unit (ALU) was previously loaded from the RAM and not the hard drive directly. Thus, the RAM is a scarce resource that should be precisely, efficiently and carefully managed by the OS in order to achieve the best performance possible. Only vital data should be kept in the RAM; the rest should be kept in secondary memory.

Programs that quit should return the memory that they had been allocated while running. The OS should guarantee that this will happen, because otherwise those resources won't be available to any other programs. This situation is rare and is called a **memory leak** or **resource leak**. The OS should dynamically allocate portions of memory to applications at their request and also free them for reuse when no longer needed by the application. This is crucial since, in modern computer systems, a lot of processes are running at any one point in time.

During the operation of any computer system, the OS should track the location of the programs within its RAM and convert the **logical** into **actual physical addresses**, whenever those addresses are needed by a process. A logical address is a reference to a stored value and is used by the program that generated this reference, while a physical address is the actual address of one memory cell. **Address binding** is the process of mapping a logical address to a physical address. A program may be loaded in different physical addresses, while running, and the use of address binding helps to keep track of where the program is located in RAM. Since the logical address is known, the physical address of a process can also be located. Using these techniques the OS keeps track of programs in memory.

**Swapping** is the general term of a mechanism in which a process or blocks of program code can be swapped temporarily out of RAM and into a hard disk, and then, later, brought back into RAM to continue their execution.

**Paging** is the underlying mechanism of **virtual memory** implementation, which allows all modern OS to utilize the, much greater in size, secondary storage as if it was RAM. The OS copies as much data as possible into RAM, and leaves the rest on the disk. When the OS requests data from the disk, it exchanges a quota of data (called a **page**) in RAM with a quota of data on the disk. This is extremely helpful when data to be loaded in RAM is greater in size than the available RAM. The secondary storage used in this case is usually a hard disk drive capable of providing direct access to these memory pages. Excessive page swapping causes **thrashing** that results to poor system performance.

Multitasking systems use **slicing** to effectively manage all running programs. A **slice** or a time-slice is the time allocated to each user in a multi-access system or to a program in a multitasking system. The OS uses an **interrupt mechanism** which suspends a process that is executed by the ALU and invokes a mechanism to identify the next process to be executed. The **interrupt handler** is scheduled to allow the OS to switch between processes when their time-slices expire. The mechanism used to select the next process is called a **scheduler** and

is run once, during every time-slice, to choose the next process to run. The collaboration between the interrupt and the scheduler allows for the processor's time to be shared between a number of different processes and is a vital component of multitasking systems. In a modern multitasking OS, the operating system can store and restore the state of a process or thread, so that execution can be resumed from the same point at a later time. The scheduler performs selections that satisfy the scheduling policy's priority constraint. When a high priority takes over from the lower priority task currently running, it is known as **preemptive scheduling**. When the scheduler selects a ceased lower priority task to resume, the task continues from the stored state.

At any specific time, processes can be separated into two groups: those that are waiting for input or output and those that need to use the CPU. The processes that are waiting for input or output do not need to use the CPU, allowing other processes to do so. When the requested data becomes available, and thus the input or output has completed, an interrupt is generated and the paused processes return to their executing phases and may use the CPU.

*Managing peripherals*

Input and output devices vary in their characteristics, come from different vendors and present different technical challenges. Their speed of communication also varies, even when they belong to the same industry standard. For example, USB 1.0, released in 1996, supported data rates of 1.5 Mbit/s to 12 Mbit/s and USB 1.1, released in 1998, fixed problems identified in 1.0. USB 2.0, released in 2000, supported data rates of 280 Mbit/s, while USB 3.0, released in 2008, supported data rates of 4 Gbit/s. Finally, USB 3.1, released in 2013, supports data rates up to 10 Gbit/s. The OS uses a special program called a **device driver** to handle all internal and external hardware, input and output peripherals and storage devices. Each device connected to a computer system utilizes a device driver in order to communicate with the OS. This piece of software acts like a bridge that facilitates the communication between the particular piece of hardware and the operating system. A device driver is coded by the product manufacturer and guides the OS as to how to use the particular device. Manufacturers usually provide free up-to-date device drivers and most device drivers are operating-system-specific. The use of device drivers is another example of abstraction since the OS does not need to know the technical details about every piece of hardware that needs to exchange data.

*Managing hardware interfaces*

The operating system hides the complexity of hardware resources from users while it also manages the interaction of processors, memory, data storage and I/O devices. It acts like an interface that handles "interrupts" generated by the I/O controllers and shares I/O between programs using the CPU. This is also is another example of abstraction. The OS efficiently manages low level hardware in a way that application software can take advantage of installed hardware. For example, when typing a document, the word processor application does not bother with the drivers of the keyboard or the video adapter used. That is an example of abstraction. The OS multiplexes the hardware components for all application software and hides all unnecessary details from the user and the application software.

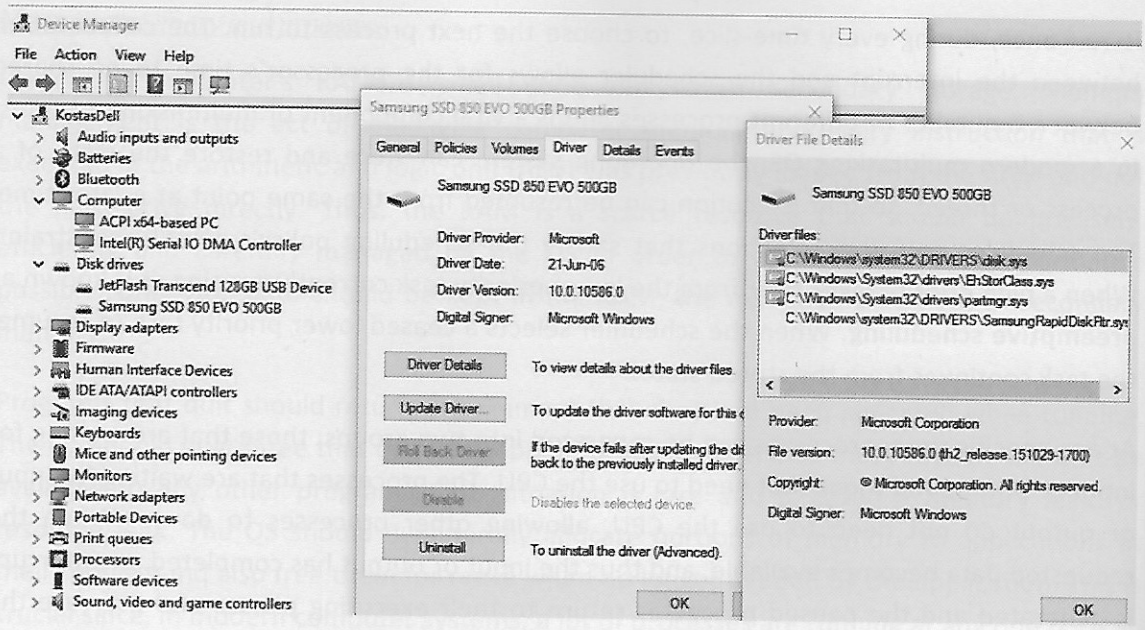



**Figure 6.11: The device manager of Windows 10**

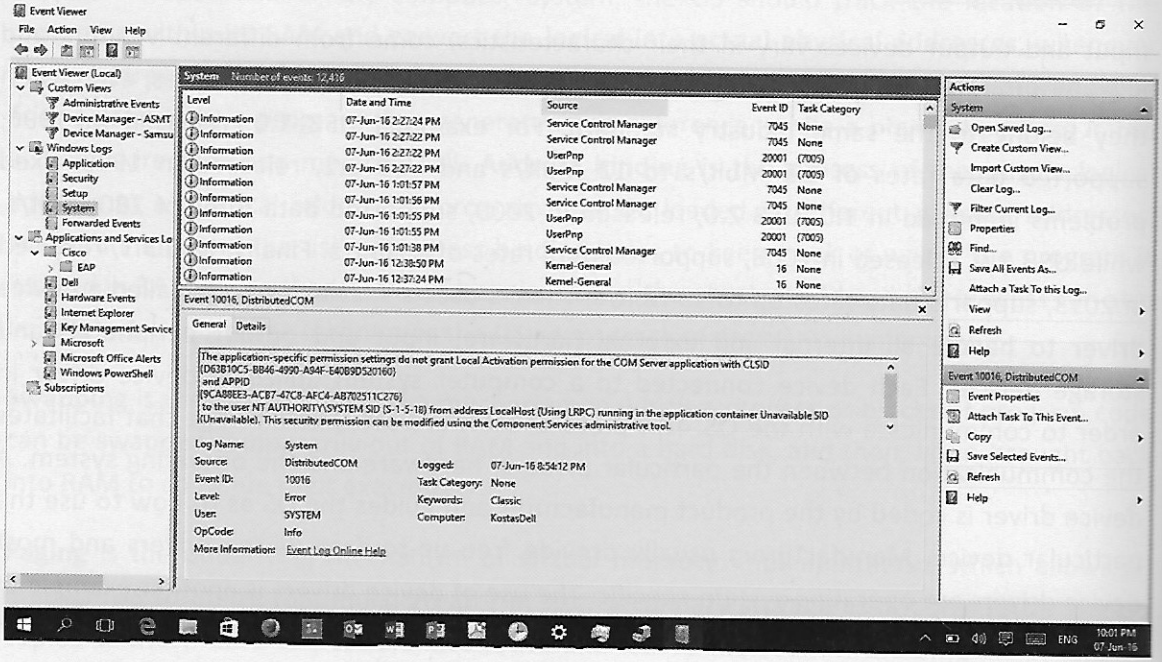



**Figure 6.12: Event viewer of Windows 10.**

## 6.1.6 – 6.1.7 OS resource management techniques

| Exit skills. Students should be able to[1]: |
| --- |
| Outline OS resource management techniques: scheduling, policies, multitasking, virtual memory, paging, interrupt, polling. |

The techniques of multitasking, virtual memory and paging have been discussed in the previous section.

*Scheduling*

Most OSs have a special application called "**Task Scheduler**" that is used to create and manage common tasks that the computer should carry out at specific times. These tasks include virus scans, backups, defragmentation etc.

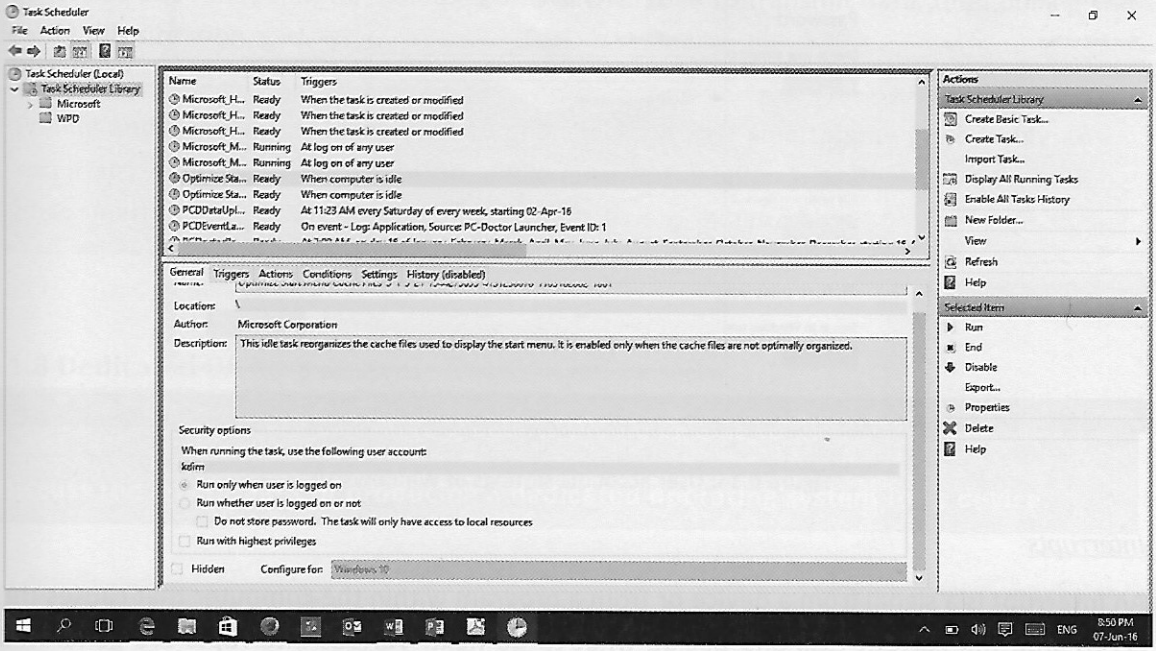



**Figure 6.13. The Task Scheduler of Windows 10**

The **process scheduler** is part of the operating system and decides on the next tasks to be admitted into the CPU and thus the next process to run. Operating systems may feature up to three separate scheduler types (short-term, medium-term and long-term). The scheduling criteria that affect the selection of the best scheduling algorithm for a particular situation include CPU utilization, throughput, turnaround time, waiting time and response time. The most common CPU scheduling algorithms are:

1. First-Come First-Serve Scheduling, FCFS
2. Shortest-Job-First Scheduling, SJF
3. Priority Scheduling
4. Round Robin Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback-Queue Scheduling

*Policies and account management*

In many cases more than one user has access to a personal computer. Moreover, in a corporate computer network, there could be thousands of people who have access to the network's data. In all cases, an operating system is responsible for setting up accounts for each user who will utilize the computer or network resources. A user account defines the privileges and access rights of a particular user. A user name and a password is used to gain legitimate access to a computer or a network and to prevent unauthorized access by people who do not have the permission to use the services and the facilities.

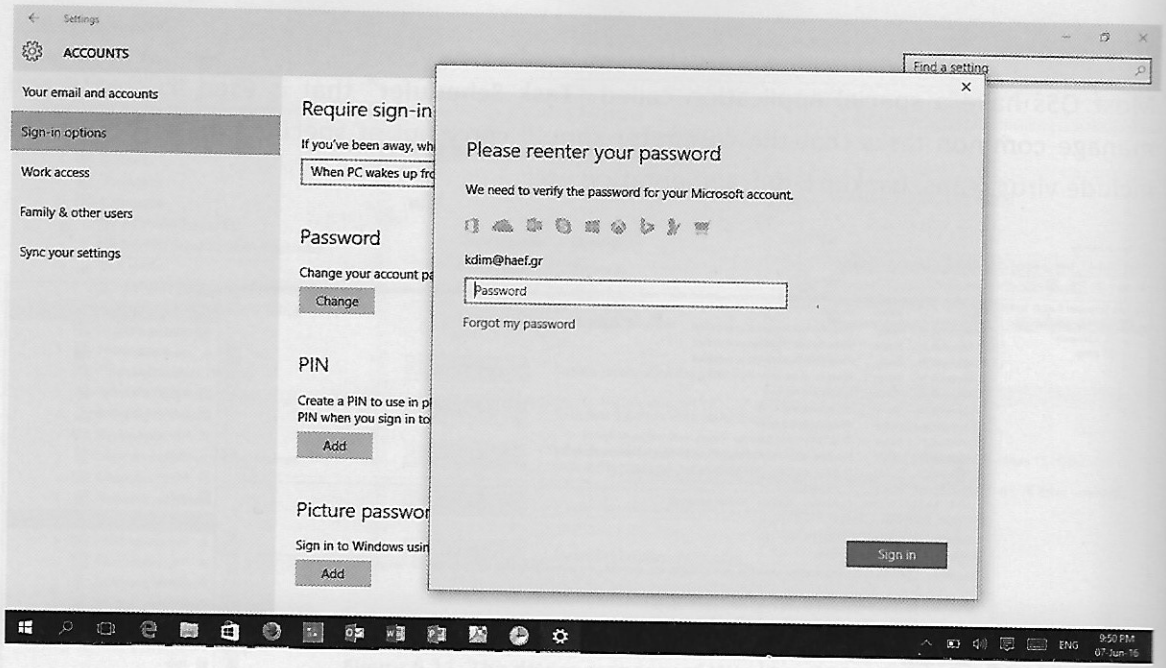



**Figure 6.14: User accounts settings of Windows 10**

*Interrupts*

An **interrupt** is a signal from a device or from a program within the computer that causes the OS to stop the current task and decide what to do next. Various interrupts are generated and have different priorities. When a printer runs out of paper a "printer unavailable" hardware interrupt will be triggered to inform users that the printer is not available for printing. The multitasking principle, described in this chapter, is based on software interrupts. In general a hardware interrupt occurs when an I/O operation has finished performing a function, while a software interrupt occurs when a program requests various services from the OS.

Task Manager

File Options View

Processes | Performance | App history | Startup | Users | Details | Services

| Name | 6% CPU | 62% Memory | 1% Disk | 0% Network |
|---|---|---|---|---|
| Microsoft account | 3.1% | 56.8 MB | 0 MB/s | 0 Mbps |
| Desktop Window Manager | 1.1% | 67.9 MB | 0 MB/s | 0 Mbps |
| System and compressed memory | 0.9% | 0.1 MB | 0.1 MB/s | 0 Mbps |
| Task Manager | 0.5% | 19.5 MB | 0 MB/s | 0 Mbps |
| Store | 0.4% | 40.4 MB | 0 MB/s | 0 Mbps |
| System interrupts | 0.1% | 0 MB | 0 MB/s | 0 Mbps |
| WMI Provider Host | 0% | 24.4 MB | 0 MB/s | 0 Mbps |

**Figure 6.15: Various system interupts presented in the Task Manager (Windows 10)**

*Polling*

**Polling** is the periodic checking of devices, by a central device to sample their status (i.e. see what state they are in and identify whether they are still connected or want to exchange data). The CPU periodically checks certain registers, actuators or sensors to see if some request has been made. If, for example, a device is ready to transmit data, then polling will identify this situation.

**Polling and interrupts:** Polling is when one checks his/her smartphone periodically, to see if any notification has come up, while interrupt is when a notification arrives to ones smartphone and the smartphone vibrates to inform him/her.

### 6.1.8 Dedicated OS for a device

| Exit skills. Students should be able to[1]: - |
| --- |
| Discuss the advantages of producing a dedicated operating system for a device. |

There are two general approaches when developing a dedicated OS for a device:

- The first approach is to take an existing OS and adapt it for the particular device. The advantage of this approach is that the end user deals with a familiar interface. The disadvantage is that the final product will not be optimized for this particular purpose. For example, if one plans to develop an OS for a mobile device, using an existing OS that was developed for desktop computers, as its base, this may lead to various obstacles since the mobile device may not have all the resources and capabilities of a desktop computer.
- The second is to design an OS that will fit the particular needs of the device exactly. This approach would lead to an OS optimized for this particular device and purpose. For example, when developing an OS for an embedded system, characteristics such as small size (in bytes), quick responsiveness to external interrupts, real-time scheduling policy and fast, lightweight, processes are very important and should be taken into consideration. When developing an OS for a cell phone, considerations about the efficient use of the battery and RAM, as well as the small touchscreen size and the overall small device size should be made.

**Examples of dedicated OSs:**

**Android** is a the name of the mobile Operating System currently developed and owned by Google. It is based on the Linux kernel, and designed mainly for touchscreen mobile devices, such as tablets and smartphones. Android is one of the best-selling OSs for handheld devices, and has the largest installed base. It is written in C, C++ and Java, and offers users access to Google's own services like Google Search, Maps, Translate, YouTube and Gmail.




**Figure 6.16: Android Nougat replica in front of Google office on June 30, 2016.**

**Symbian** is a mobile OS designed for smartphones. It was written in C++ and originally developed as a closed-source OS for PDAs in 1998 by Symbian Ltd. Symbian was used by many mobile phone brands, like Samsung, Sony Ericsson, and Nokia, but is defunct as of May 2014.

**TinyOS** is a free and open-source, embedded, component-based, OS and platform for low-power wireless devices, such as those used in wireless sensor networks and home automation. It is written in the nesC programming language.

**Tizen** is an OS based on the Linux kernel and the GNU C Library implementing the Linux API. It is written in C, C++ and HTML, and is a project within the Linux Foundation. The Technical Steering Group that governs this project include Samsung and Intel. It works on smartphones, PCs, cameras, tablets, in-vehicle infotainment devices, smart TVs, smartwatches, Blu-ray players, printers, refrigerators, air conditioners, etc. Its purpose is to offer a consistent user experience across devices and promote smart home solutions.

**Embedded Configurable Operating System (eCos)** is a free and open-source, real-time, OS that was engineered to serve embedded systems and applications which need only one process with multiple threads. It is easily customizable to specific application specifications of run-time performance and hardware requirements. It is written in C and C++.

### 6.1.9 OS and complexity hiding

| Exit skills. Students should be able to[1]: |
| --- |
| Outline how an operating system hides the complexity of the hardware from users and applications. |

As we have already seen earlier in this chapter the OS hides the complexity of the hardware from the users. This is a typical example of abstraction. The use of device drivers and the management of hardware resources are not the only examples of abstraction related to OS function.

*Drive letters*

In the following picture, drive C is a SSHD and D is a USB device. So letters C and D correspond to real devices. Desktop, documents, downloads, 101__05, BOOK etc. on the other hand refer to folders (locations). The user uses the icons to select the desired folder without bothering with the complexity that lies beneath this simple selection (Figure 6.17).

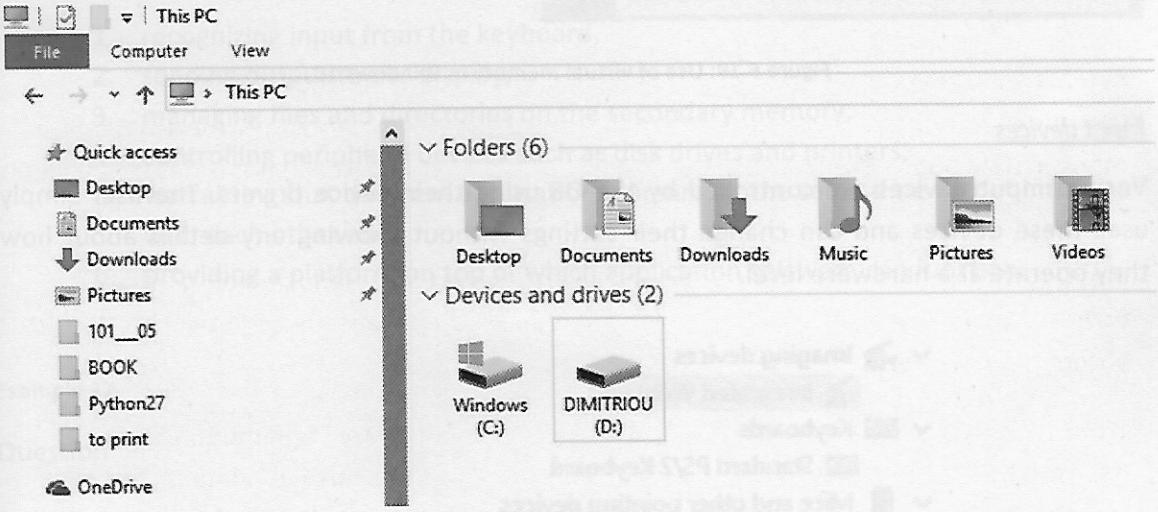



**Figure 6.17: File explorer of Windows 10.**

*Virtual memory*

A user of Microsoft's Windows 10 has the capability to alter the settings of virtual memory. After that, the OS will handle the page file, when required, in the best possible manner, in order to maximize the performance of the system. Once again the OS hides this complexity.

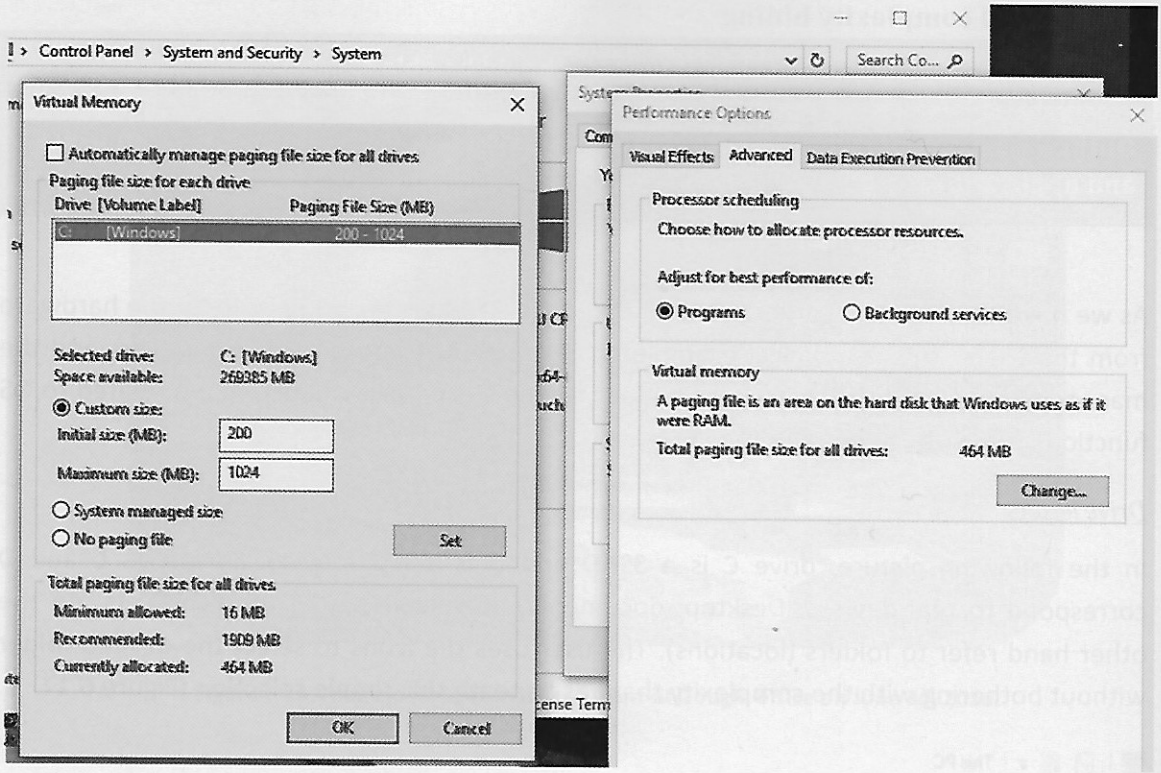



**Figure 6.18: Use of virtual memory in Windows 10**

*Input devices*

Various input devices are controlled by the OS using their device drivers. The user simply uses these devices and can change their settings without knowing any details about how they operate at a hardware level.

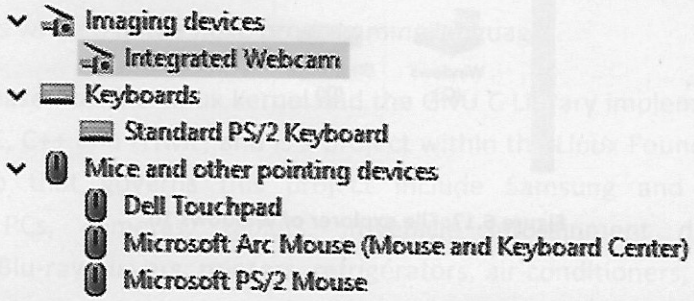



**Figure 6.19: Screenshot showing various input devices in the device manager**

*The Java virtual machine*

The Java architecture allows code to run on any machine on which the Java Virtual Machine interpreter has been installed. In Java architecture, all details of making the code function on a specific hardware platform are handled by the Java Virtual Machine (JVM). This is another example of abstraction.

## End of chapter example questions with answers

**Example 25**

Question

What are the two main categories of software?

Answer

Application and system software.

**Example 26**

Question

State some main tasks of the operating system.

Answer

1. recognizing input from the keyboard,
2. sending output to the display,
3. managing files and directories on the secondary memory,
4. controlling peripheral devices such as disk drives and printers,
5. warranting that different applications running at the same time do not interfere with each other and
6. providing a platform on top of which application software can function.

**Example 27**

Question

State two types of operating system strategies?

Answer

Batch and timesharing.

**Example 28**

Question

What is a resource in a computer system?

Answer

A resource is anything that can be allocated and managed by a computer system.

**Example 29**

Question

State some examples of resources.

Answer

CPUs, input/output devices and RAM.

**Example 30**

Question

Explain why the problem of resource allocation is complex?

Answer

Modern computers run several applications simultaneously. All these applications need to compete for the limited computer resources available, thus producing a complex problem of resource allocation.

**Example 31**

Question

Describe an example of abstraction relating to resource allocation.

Answer

When allocating resources, all details of how the hardware operates are hidden. The programmer writes an application without bothering with all these details.

**Example 32**

Question

What is the difference between a program and a process?

Answer

Although the terms are used interchangeably a process is basically a program in execution.

**Example 33**

Question

What is the process life cycle?

Answer

When a process executes, it passes through start, ready state, waiting state, running state and terminated.

**Example 34**

Question

Explain what is a running process, a ready process and a waiting process.

Answer

The running state means that a process has all the resources it needs for execution and the operating system has given it permission to use the processor and execute its instructions.

The ready state means a process has all the resources it needs, but is waiting to be assigned to a processor.

The waiting state means a process is waiting for a resource, such as input from the user or a file to become available.

**Example 35**

Question

What is deadlock?

Answer

Deadlock is a situation that can arise when two processes hold resources and request other resources, from each other, at the same time. Process A holds a resource that process B wants, while process A requests a resource that process B holds. The result is that neither can continue.

**Example 36**

Question

What is fragmentation?

Answer

Fragmentation is a phenomenon or state in which memory is used inefficiently and is broken up in small pieces. This leads to files and programs being divided up and stored in various areas throughout the memory, which results to slower computer systems, when these files and programs need to be accessed. This happens because processes, which occupy different amounts of memory, are constantly allocated and deallocated to and from the memory.

**Example 37**

Question

Explain how a PC can run processes which are larger than the available RAM?

Answer

The system can use secondary storage as if it was primary memory. This is known as virtual memory.

**Example 38**

Question

What is the difference between mainframe computers and supercomputers?

Answer

A supercomputer has as a dedicated purpose to run demanding programs (e.g. scientific research or engineering models). It holds great processing power in order to execute a program as quickly as possible. It always runs at maximum performance, solving a particular problem. Its performance is measured in Floating Point Operations per Second (FLOPS).

A mainframe typically runs multiple programs concurrently requested by many concurrent users. It processes huge amounts of external data and its performance is measured in Millions of Instructions per Second (MIPS).

**Example 39**

Question

What is the file manager of an OS?

Answer

The file manager is responsible for the management and maintenance of the secondary memory (e.g.: USBs, SSHDs, hard disks). It provides:

- folders and directories to organize files,
- commands to create and delete folders and directories,
- commands to read and write to a file,
- commands to set protection to a file,
- commands to set and change the ownership of a file or folder,
- commands to delete a file.

**Example 40**

Question

Describe the difference between physical primary memory, virtual memory, and logical memory.

Answer

Physical primary memory is the memory available by a computer to execute processes. Virtual memory is a technique through which applications that require space larger than the available RAM can be executed by using disk memory as if it was primary memory. Logical memory is an abstraction of the computer's different types of memory that facilitates programming.

**Example 41**

Question

Why is RAM not suitable for permanent program storage or backup purposes?

Answer

RAM is a volatile memory, while disk drives can store data permanently.

**Example 42**

Question

State one advantage of multi-core processing over multi-processor systems.

Answer

Communication between processors on the same chip is faster than processors on separate chips.

**Example 43**

Question

Why is a high level language suitable to implement an operating system?

Answer

The code can be written faster and is easier to understand and maintain.

## Chapter References


1. International Baccalaureate Organization. (2012). IBDP Computer Science Guide.
2. IBM sets new tape storage record. (13, April, 2015). In *New Atlas*. Retrieved 19:05, June 13, 2016, from http://www.gizmag.com/ibm-tape-storage-record/36931/
3. The digital revolution demands a better server. In *IBM*. Retrieved 20:05, June 14, 2016, from http://www-03.ibm.com/systems/z/hardware/z13.html
4. Arnheim, R. (1969). Visual Thinking. University of California Press, USA.
5. Bishop, P. (1987). Computer science. Nelson. Australia.
6. Bradley, R. (1999). Understanding computer science for advanced level. Nelson Thores, UK.
7. Brookshear, J. (2012). Computer science an overview. Addison-Wesley. USA.
8. Dale, N. and J. Lewis. (2013). Computer science illuminated. Jones & Barlett Learning, UK.
9. Sanders, D. (1988). Computers today. McGraw-Hill, Singapore.
10. Silver, G. & Myrna Silver. (1993). Computers and information processing. HarperCollins, USA.
11. Willis, N. & John Kerridge. (1983). Introduction to computer architecture. Pitman Publishing, UK.
12. Wise, H. (1981). Computer Architecture. Blackie, UK.

# TOPIC 7 – Control

## Topic 7 — Control[1]

## 7.1 Control

### Centralized control systems

### 7.1.1 A range of control systems

| Exit skills. Students should be able to[1]: |
| --- |
| Discuss a range of control systems. |

A control system[2] is one or more devices that guide other devices or systems. This allows for the completion of various tasks in an automatic manner, without any human intervention. Control systems are set up once and can then perform the actions that they have been programmed to do in an automatic manner, relieving humans from repetitive and mundane tasks, as well as increasing productivity and efficiency. Control systems may include sensors to gain feedback from the environment and motors to control any actuators in an appropriate manner.



**Image 7.1: Automatic doors**

An example of a control system is contained in the automatic doors that can be found in various locations, such as supermarkets. Instead of humans manually having to open and close these doors, a motion sensor is located at the top of the automatic doors, which allows the control system to become aware of any motion. The sensor may use infrared or microwave signals that bounce off of objects to determine motion. After receiving feedback from the environment, the sensor forwards its signal to a microprocessor and allows the control system to operate its actuators to open the doors only when motion is detected. Furthermore, more advanced automatic doors include photocell sensors that can tell if

[1] International Baccalaureate Organization. (2012). IBDP Computer Science Guide.
[2] Control System. (12, May 2016). In *Wikipedia, The Free Encyclopedia*. Retrieved 19:05, May 13, 2016, from https://en.wikipedia.org/wiki/Control_system

someone is at the door, so that the doors do not close on them. This results in a control system that is very convenient, as well as efficient, when one considers that doors cannot now be left open accidentally, which may be costly when a heating or air-conditioning system is operating. Furthermore, automatic doors allow access to both the elderly and the disabled, since both might find it difficult to open or close a manual door.

Control systems are mostly input, process, output systems. That means that an input is provided to the system, which is processed by some algorithm, and an action is performed. The results of the action can then be measured by the control system (through the use of sensors) and feedback can be provided as input. The control system can then act upon the new input and the cycle continues as seen in Figure 7.1.

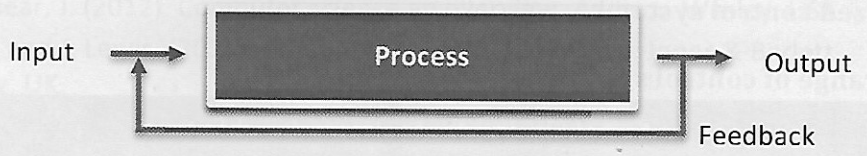



**Figure 7.1: Control System**

There are some circumstances however where an action might need to take place while the control system is performing some other action. For example, when a second person approaches an automatic door while it is closing, a signal must be sent to the control system to re-open the doors. In this situation the signal sent to the control system is known as an *interrupt*. An interrupt is a signal sent to the control system indicating that the system needs to attend to the specific signal immediately.

A number of control systems are considered below. All of them use computer systems in order to operate and so depict the many possibilities for control systems with developments in computer systems.

- **Heating system**: In any such control system, be it in a car, a house or any other place, an initial temperature is given as input by the user (eg. 24°C). That input signifies the ideal value of the output and the goal of the control system. That is, the control system will aim to reach the value provided by the user. Sensors allow the control system to measure the temperature of the environment and determine any actions that it might need to perform, in order to reach the desired temperature. For example, given an actual temperature of 18°C (as input from the sensors) and a desired temperature of 24°C (as input from the user), the control system would turn on the heater until the input from the temperature sensors was equal to the desired user temperature. At that point, the heater would turn off and would turn on again only

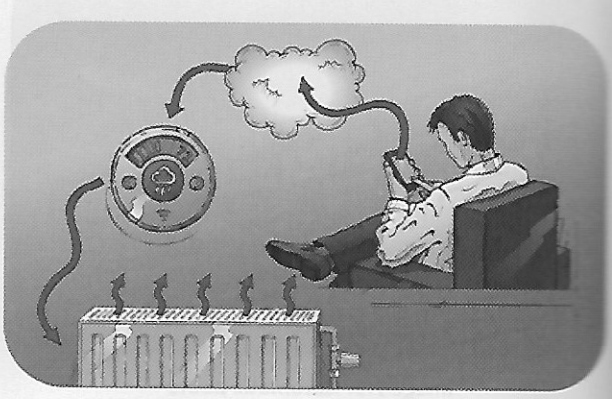


**Image 7.2: Smart heating system**

when the actual temperature falls below the desired temperature. Having such a control system allows for the conservation of energy and is more efficient than turning a heating system on and off manually. This is because the automatic system would reach the desired temperature once and then turn on to compensate for even a small drop in the desired temperature. On the other hand, in a manual system, since there would not usually be a user holding a thermometer and turning the system on and off, every time the temperature fell or rose above the desired temperature, the energy wasted would be greater. Furthermore, newer "smart"[3] heating systems also have the potential to connect to the Internet, allowing the user to operate them from afar. For example, one could turn on the heating system of his/her house as one leaves work, so that the temperature is ideal when the user arrives home.

- **Taxi meters**: Initially, taxi meters were mechanical, bulky and made a characteristic ticking sound. Throughout the years, and through the use of computer-operated control systems, taximeters have improved, preventing fraud as well as providing a number of helpful features. For example, receipts are automatically issued when a fare is completed. Seat sensors detect the presence of passengers in order to prevent passenger journeys that do not operate the taximeter. Credit cards are supported, as well as radio communication and GPS systems, so to assist drivers, provide security, and better calculate distance. Taxi meters have become more efficient, less error-prone, and automatic, relieving the driver from hassles that could distract him/her from driving.

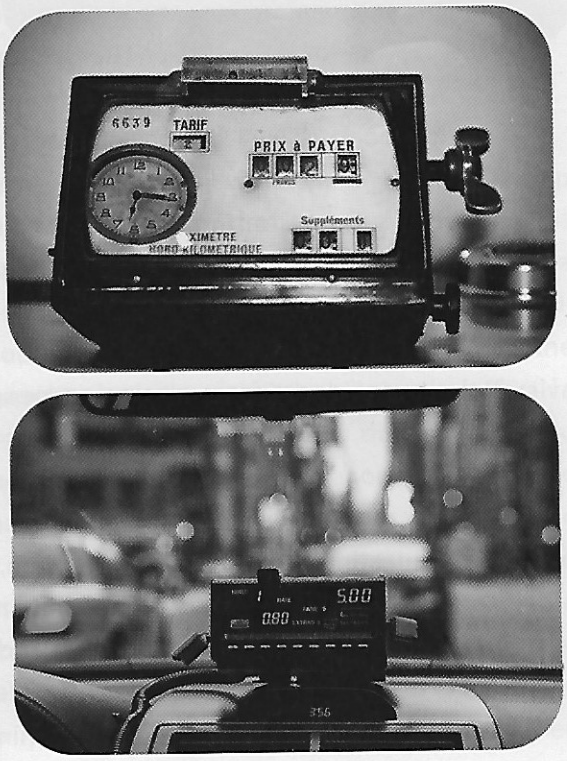



**Image 7.3: Old and new taxi meters**

- **Elevators**: Primitive elevators date back to ancient times and were notoriously dangerous and mostly used to move objects, instead of people. As engineering improved, so did elevator technology.

[3] Smart thermostat. (19, April 2016). In *Wikipedia, The Free Encyclopedia*. Retrieved 19:25, May 13, 2016, https://en.wikipedia.org/wiki/Smart_thermostat

Developments in computer systems also played a significant role and many modern elevators are controlled by computer operated control systems. An elevator control system takes in a number of sensor readings as well as the desired destination, in order to turn a winch motor with the correct number of revolutions so that the elevator car reaches its destination. The system needs to know the initial and the desired destinations, as well as the location of each floor. The system reads the desired destination from the user and uses sensors to detect the current location, as well as the location of each floor. In many systems, a load (weight) sensor may also exist that tells the control system how full the car is, so that the computer system in control may use this information accordingly (ex. not stop to take more passengers without unloading some current passengers). Finally, elevators also include automatic doors that open and close only when the elevator car is at specific floors. They also use motion sensors, as discussed before, to prevent doors from closing on people entering or exiting the car. Many elevators include a stop button, that stops the elevator car from moving in emergency situations (for example, when a dress is caught between the elevator doors). When pressed, this button sends an interrupt signal to the processor of the control system, requesting its immediate attention, and stopping the car. As such, even though elevators are control systems that are being used every day, efficiently and securely, a number of inputs from sensors and complex algorithms are constantly executed by control systems to make this possible.

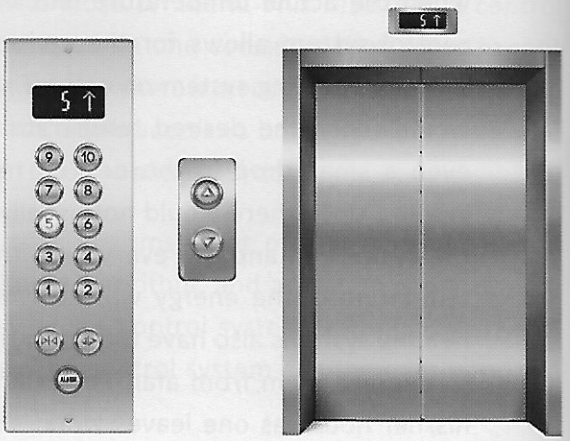


Image 7.4: Elevator

- **Washing machines**: The first washing machines date back to the mid 18th century, when manually rotating drum washers were first introduced. Washing machines with electro-mechanical timers appeared in the mid 20th century, making automatic washing possible. In the late 20th century, washing machines with control systems driven by computer systems were first introduced. In modern washing machines, sensors are used to determine and control the load size, water level, temperature, as well as the user interface. All these sensors allow for a finely-tuned washing cycle that better suits the needs of the clothes, as well as take into account the environment by consuming less electricity and water. Furthermore, modern washing machines are safer, since the door is controlled by a computer



Image 7.5: Washing machine

system that locks it during operation, avoiding accidents. If the door is opened before the washing machine has finished, or if the stop button is pressed, an interrupt signal is sent to the processor of the control system, requesting its immediate attention, and stopping the washing machine.

- **Process control**: Process control refers to the control of one or more variables (such as temperature) in a system. It strives to maintain the output of a specific process within a desired range. **Open** and **closed** loop (feedback) controllers exist. An example of an open loop controller is a tank with a manual valve that controls the heat dissipated by a coil used to heat the liquid in the tank. In this scenario, there is no feedback as to the temperature of the liquid at any one point in time. Only if the relationship between the heat dissipated by the coil and the temperature increase of the liquid is known, can one be certain of the liquid's temperature. On the other hand, in a closed loop (feedback) controller, the actual temperature of the liquid is detected, using temperature sensors, and the coil temperature is adjusted to achieve the desired liquid temperature. In this example, the valve can either be an on/off valve or a variable one. With an on/off valve the valve is turned on until the desired liquid temperature was reached, then turned off, and on again after the temperature dropped below the desired temperature. With a variable valve, the heat through the coil is proportional to the difference between the current and the desired liquid temperatures. The coil initially has a higher temperature, that slowly lowers, as the liquid temperature reached the desired temperature. Developments in computer systems allow for the automatic adjustments of the valve in order to achieve the desired temperature, with minimal or no human interaction. If an error happened with the valve so that it did not work as expected (for example, it did not turn off and the desired liquid temperature was exceeded by an undesirable amount), an interrupt could be sent to the processor, indicating that the processor would need to bring the system to a hold immediately.

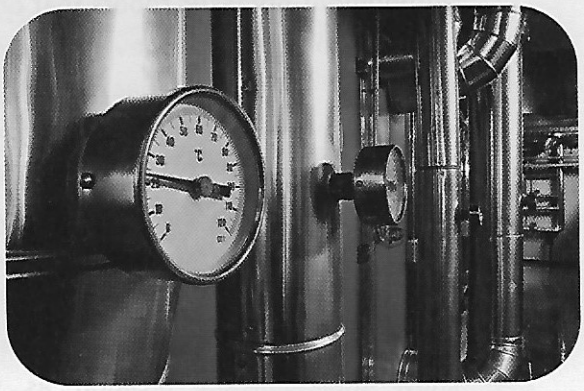


**Image 7.6: Tank with temperature gauge**

- **Device drivers**: Device drivers are computer programs used to control a computer device of any kind, from keyboards and printers to graphics and sound cards. These computer programs allow operating systems (OS) and other programs to access the devices with a layer of abstraction (i.e. without needing to know how the actual devices are used). The computer bus usually connects the drivers with the devices, allowing the drivers to send commands and receive data. Since drivers are developed specifically for each device, they are hardware-dependent, as well as operating-system specific, so that they are as efficient as possible.

- **Domestic robots**: Domestic robots using computer guided control systems are already appearing in households. Vacuum, pool and gutter cleaning robots are examples of autonomous domestic robots that one might come in contact with in everyday houses. A number of sensors are packed within these robots to provide "intelligent" behavior. For example, vacuum cleaning robots use a suite of sensors to map and find their way around rooms that may include clutter and furniture. Using various cameras they can create house maps, using landmarks, and know their current position as well as previous locations. Furthermore, they can return to their charging base whenever necessary, and complete the cleaning afterwards. Sensors can also detect the material on which the vacuum is currently on and perform appropriate kinds of cleaning. All these possibilities would not be available without closed-loop computer-controlled systems.



**Image 7.7: Cleaning robot**

- **GPS systems**: Every day millions of people use GPS systems to find their way around cities, mountains or even the sea. Whether used to find a new café or guide a missile, GPS systems are required to work efficiently, precisely and with near 100% availability (no down time). The GPS system is a large, complex, computer controlled system that includes both hardware and software elements and needs constant maintenance and support to work in such an effective manner as it does. It uses around 30 satellites in orbit around the Earth, as well as a number of extra satellites that are "fail-safes" (if one of the satellites fails, one of the extra satellites will take its place until the first one is fixed). These solar-powered satellites circle the Earth twice per day. A GPS receiver allows a user to know exactly where he/she is on Earth. It includes a sensor that locates four or more of the GPS satellites, figures out its distance to each one, calculates the time difference between the satelites and uses a mathematical principle called trilateration to deduce its location. Furthermore, modern GPS systems also receive input from their users, pinpointing their desired destinations, as well as map data that may be updated through software updates.

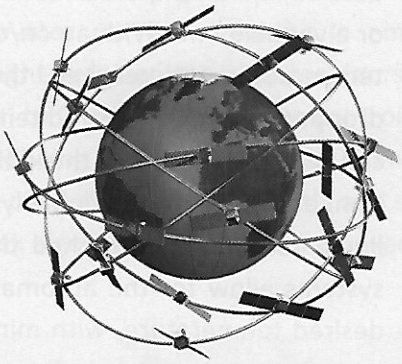


**Image 7.8: GPS satellite system**

- **Traffic lights**: Traffic lights play a key role in road safety. They manage traffic, prevent accidents and allow pedestrians to cross the roads without jeopardizing their lives. Traffic lights are usually control systems operated by computer systems and can be either "fixed time" or "dynamic control".
    - **Fixed time**: As the name suggests, fixed time traffic lights are open loop control systems, in that they do not receive any feedback from the environment in order to alter their performance. They are configured to change color after a given period of time and they do so repeatedly. Fixed time traffic lights can be implemented with an electro-mechanical signal controller and as such does not necessarily need a computer system.



Image 7.9: Traffic light with sensor

    - **Dynamic control**: Using a closed loop control system to provide feedback on the amount of traffic passing by, dynamic control traffic lights can adapt their settings appropriately, following some algorithm. These kinds of traffic lights are operated through computer systems and would not be available otherwise. They can use one of the two different types of sensors to detect traffic conditions:
      either embedded into the surface of the road or mounted on the traffic light itself (or some other high position). The sensors that are embedded into the surface of the road have the ability to sense when a car passes over it. Mounted sensors are less expensive and can provide the same feedback, including real-time photos or video.

### 7.1.2 The uses of microprocessors and sensor input in control systems

| Exit skills. Students should be able to[1]: |
| --- |
| These should be related to the examples suggested above. |

Today, most modern control systems use microprocessors in order to efficiently and effectively read sensor input, process it according to an algorithm, and finally perform some action(s).

A microprocessor, as its name suggests, is a "small" processor, that contains most, or all, of a central processing unit (CPU) functions on a single chip or integrated circuit (IC). Like a CPU, a microprocessor performs arithmetic and logic operations, as well as any other data

operations necessary, through the use of registers. This is described in detail in Topic 2 – Computer Organization in the Core Computer Science for the IB Diploma Program book[4]. Therefore, it is a programmable, input-process-output device that also includes some internal memory.

Various types of microprocessors exist, all of which have been developed with some specific purpose in mind:

- **General purpose:** Capable of running a wide range of different programs and usually integrated into a larger system. The CPU found in a desktop computer is a general purpose microprocessor. Desktop computers also include various peripheral devices, as well as external memory, all of which the microprocessor needs to interact with.
- **Embedded controller or microcontroller:** Microcontrollers are usually stand-alone chips that may include the main elements of a larger system, such as RAM and ROM, although smaller in capacity. These controllers are designed to perform some precise task(s) and do not need, or cannot include, a whole computer system. Microcontrollers may be found in most of the control systems described in section 7.1.1. Automatic doors, heating systems, taxi meters, elevators, washing machines, domestic robots, GPS systems and traffic lights, all use microcontrollers to function in an efficient and timely manner. Microcontrollers, in general, need less power to function and are smaller in size, compared to desktop computers. They can be integrated into smaller, lower powered electronic devices.
- **Graphics processing unit (GPU):** Computer graphics have become so detailed and realistic in recent decades that an additional, special kind of microprocessor, is included in most computer systems to handle them. This type of microprocessor is known as the GPU and includes hardware to allow for faster handling of graphics related mathematics (such as matrix multiplication and vector arithmetic). GPUs are responsible for calculating and rendering polygons and pixels on the screen.

Microprocessors improve control systems in a number of ways. First of all, they can process input data much faster than a human ever could and as such they can react to changes in the input quickly. Control systems that depend on microprocessors are automated and as such are error-free compared to manual or mechanical systems. Furthermore, they can operate throughout the year with little to no off-time (down time) and in conditions that are harsh or dangerous for humans.

Of course, since embedded microprocessors are pre-programmed systems that follow some specific algorithms, they would probably not be able to operate in the most effective way or at all (in an unexpected event). Moreover, as microprocessors need power to operate, the system would not be able to function if there was a power shortage.

[4] Dimitriou K. Hatzitaskos M. (2015). Core Computer Science for the IB Diploma Program. Athens: Express Publishing. More information at: https://www.expresspublishing.co.uk/gr/en/content/core-computer-science-ib-diploma-program

In order for microprocessors to perform any processing, they need to receive input. For that to be accomplished sensor input is needed. A sensor converts continuous physical (analogue) quantities (such as speed, temperature, humidity, pressure, etc.) into discrete digital signals using an analogue-to-digital converter (ADC). These signals can then be read as input by microprocessors. The input is processed, according to some algorithm(s), and output is returned.

Although a wide variety of sensors exist[5], all of which share a number of common properties determining their quality:

- **Accuracy:** Determines whether the measurement of the physical quantity and therefore the final delivered digital signal is accurate (for example, a temperature of 18°C should not be measured as 19°C).
- **Range:** Determines the acceptable range of the physical quantity within which the sensor may acquire readings (for example, a temperature sensor might be able to accurately operate between -30°C to 60°C, but not above or below those temperatures).
- **Resolution:** Determines the smallest increment that the sensor may detect (for example, one temperature sensor might be able to determine increments of 1°C accurately, measuring whether the temperature is 17°C or 18°C, while another might be able to determine increments of 0.1°C accurately, measuring whether the temperature is 17.0°C or 17.1°C.

Sensors should also have two more characteristics, without which their readings would be useless. Firstly, sensors should be insensitive to any other physical conditions present that could influence the reading. For example, when measuring a temperature, the measurement should not be affected by wind. Secondly, the sensors should not influence the measured property in any way. In our temperature-measuring example, the sensor should not dissipate heat into the environment and influence the temperature around the sensor.

### 7.1.3 Different input devices for the collection of data in specified situations

| Exit skills. Students should be able to[1]: |
| --- |
| Scenarios will be based on familiar situations to students. |

As described in the previous section, sensors are usually an integral part of control systems and act as input devices. A growing list of available sensors exists[6] that can measure anything from density to heat and motion to magnetic fluctuations. Sensors sense a physical property and then transform it into an electrical signal.

[5] List of sensors. (19, April 2016). In *Wikipedia, The Free Encyclopedia*. Retrieved 19:30, May 13, 2016, https://en.wikipedia.org/wiki/List_of_sensors

**Useful Information**: An Analogue to Digital Converter or ADC is used by every system that uses sensors to convert continuous analogue data to discrete digital signals handled by a microprocessor. ADCs cannot read and convert all the data that is acquired by the sensors, but rather convert analogue data from sensors every few milliseconds (depending on the sensor). This way, the data converted by the ADCs and handled by the microprocessor is actually a sample (a part) of the actual physical quantity being measured. Depending on how frequently the sampling is performed by the ADCs, a lot of information might be lost.

The main sensor categories are discussed below. This list is non-exhaustive and more categories are available[6]. Even in the categories presented, a large number of different sensors usually exist.

Sensor types may be:

- **Sound**: Sound sensors, as the name suggests, detect sound waves and are widely used in microphones.
- **Motion**: Motion sensors detect moving objects. They are widely used in security/alarm systems, as well as automated lighting control, so that the lights are only on when necessary to conserve energy.
- **Vibration**: Vibration sensors detect vibrations and are placed on surfaces that can vibrate. They are widely used in security/alarm systems (placed on windows), as well as acoustic musical instruments (instead of microphones).
- **Optical/Image**: A wide variety of optical sensors exist, depending on the physical quantity measured. Two widely-used optical sensors are the following:
    - **Active pixel sensor (APS)**: Used in almost all digital cameras, they contain an array of pixel sensors that can imprint light.
    - **Infrared (IR)**: Used to sense invisible radiant energy with longer wavelengths than those of visible light. Widely used in security/alarm systems to detect motion at night, where visibility is low or non-existent.
- **Pressure**: Pressure sensors detect pressure. They are used in a variety of settings, including touch-screen devices, as well as the automotive industry (they regulate the engine power according to the pressure on the pedals) and others.
- **Temperature**: Temperature sensors detect temperature. They are widely used within thermostats to control the temperature of a given setting.
- **Proximity**: Proximity sensors can detect the presence of nearby objects without any physical contact. They are widely used in cars to help drivers reverse or park.

### 7.1.4 The relationship between a sensor, the processor and an output transducer

| Exit skills. Students should be able to[1]: |
| --- |
| Describe the relationship between a sensor, the process and an output transducer. Technical hardware details are not expected. |

In the previous chapter, we learnt how a sensor senses continuous, physical (analogue) quantities (such as speed, temperature, humidity, pressure, etc.) as analogue signals. It then transforms these analogue signals into discrete, digital ones using an analogue-to-digital converter (ADC). These signals can then be read as input by processors. The input is processed according to some algorithm(s), and output is delivered.

A processor performs arithmetical and logical operations and is the core element of any computer system. It receives input, in electrical form, from a number of sensors (from sensors on a tracking device, such as a mouse, to motion sensors above automatic doors) and performs some sort of output (from moving a mouse cursor to opening doors).

In the process of sensing physical quantities, using sensors, and transforming them into analogue signals, a conversion takes place. The device that converts one form of energy to another is called a **transducer**. Transducers, in computer systems, are responsible for converting physical quantities (such as speed, temperature, humidity, pressure, etc.) into electrical signals, as well as vice versa. The conversion process from one form of energy to another is called **transduction**.

Sensors can be categorized as transducers, as they sense a physical quantity and transform it into another form of energy (typically an electric signal). However, other devices (apart from sensors) can be categorized as transducers. The most common of these are called **actuators** and are used during the output. Actuators are the devices responsible for moving some kind of mechanism. They are transducers in that they receive some form of energy (usually electric current) and convert it into motion. Control systems use actuators to perform some actions in an environment (for example, a mechanical motor in an automatic door is an actuator that receives electric current and converts that form of energy into kinetic energy so that the motor moves and the doors open).

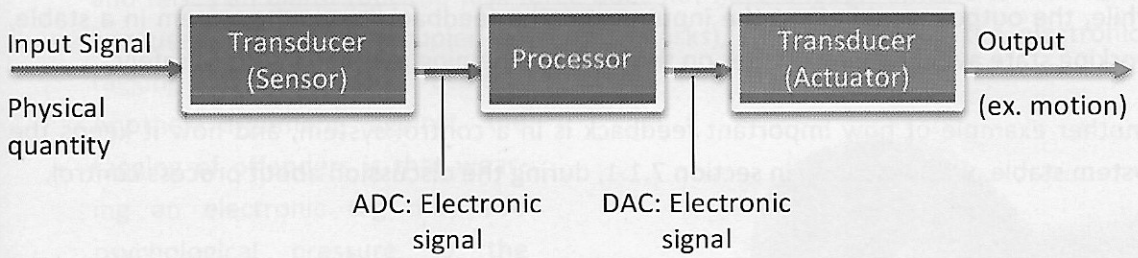



**Figure 7.2: Input, process, output and transducers**

Figure 7.2 depicts how a control system that uses a processor works as an input, process, and output device. An input signal is recorded by a sensor, which transforms the physical quantity into an electrical signal. Since it converts one form of energy to another, the sensor is a transducer. The electrical signal is further converted into an electronic signal, using an analogue-to-digital converter (ADC), so that it may be processed by the processor. After the processor runs any necessary algorithm(s) on the input data it outputs an electronic signal. That output is converted into an electrical signal, using a digital-to-analogue converter (DAC), so that it may be used as input to an actuator. The actuator will convert the electrical

signal into motion. Since the actuator converts one form of energy to another, it is a transducer.

### 7.1.5 The role of feedback in a control system

| Exit skills. Students should be able to[1]: |
| --- |
| Describe the role of feedback in a control system.<br>LINK Connecting computational thinking and program design. |

Feedback was discussed in section 7.1.1 and described in Figure 7.1. Feedback refers to the process where information about the result of an output, from a control system, is used as part of the new input to the control system in order to determine the best course of action for the next output. Feedback is a loop as shown in Figure 7.1.

Feedback is essential to control systems that need to react to their environment and its changes. For example, as discussed in section 7.1.1, there can be two kinds of traffic lights, fixed time and dynamic control. Fixed time traffic lights do not take the environment into account and just change color after a pre-programmed amount of time. On the other hand, dynamic control traffic lights use sensors to take the environment into account every time the lights change color. Imagine the following scenario: A dynamic control traffic light uses a sensor to identify when no cars are near and changes the car lights from green to red, so that the pedestrians can pass. This action is the output. The traffic light then receives feedback from this action (a line of cars starts forming) and uses that feedback as input in order to turn the car lights green again when a long line has been formed. The input first affected the output (no cars, so the car lights turned red), the output affected the input (a line formed), and the new input affected the output (the car lights turned green). After a while, the output would affect the input again. The feedback keeps the system in a stable, working state avoiding long queues on the road and helping pedestrians cross safely.

Another example of how important feedback is in a control system, and how it keeps the system stable, was described in section 7.1.1, during the discussion about process control.

### 7.1.6 Social impacts and ethical considerations associated with the use of embedded systems

| Exit skills. Students should be able to[1]: |
| --- |
| S/E For example, tagging prisoners, surveillance, CCTV, improved safety systems. |

- **Electronic tagging (tagging prisoners)**[6, 7]: Offenders that have been found guilty of minor offences, inmates that are entitled to leave time (vacation time away from the correctional center), or individuals under house arrest may be "tagged" with the use of electronic monitoring devices.

  Overcrowding of correctional facilities impacts their ability to provide effective rehabilitation. Minor offences can be efficiently dealt with using prisoner tagging. Each individual wears a bracelet or anklet that uses GPS to allow correctional officers to monitor their location at all times. Tampering with the device or trying to remove it raises an alarm.

  Electronic tagging has been shown to effectively deter crime, as well as save habitual offenders from a continued life of crime. Monitored offenders were less likely to abscond or commit new offences than those not monitored. Electronic tagging benefits society immensely by reducing crime rates, reducing the money spent on crime. There are a number of "hidden" costs surrounding any crime; for example, the time spent by police officers to solve it, instead of working to prevent possible future crimes. One should also consider court rooms and lawyer's time, as well as the occupation of prison cells and the need for the creation of further correctional facilities if the number of offenders increases.

  Electronic tagging could also improve the rehabilitation and reintegration of offenders, by allowing them to leave the correctional facilities sooner (or not enter at all), maintain gainful employment, as well as contact with their families.

  However, there is plenty of criticism concerning prisoners' electronic tagging. First of all, one could argue that it is a very lenient form of punishment. Furthermore, it does not physically restrain offenders and allows them to perform further illegal actions, if they want to. Having a large number of offenders outside the correctional facilities with electronic tags needs a large task force of individuals that can monitor them and intervene whenever necessary. If the offender performs an illegal action and raises an alarm (but the task force does not have enough officers to intervene because they are all occupied with other tasks), the idea behind the electronic tagging is rendered useless.

  Another argument against the tagging of offenders is that wearing an electronic tag may add psychological pressure to the wearer, especially to the younger offenders.

  Apart from offenders, electronic tagging could be used to monitor the whereabouts of people with health problems such as dementia (ex. Alzheimer's disease). Finally,

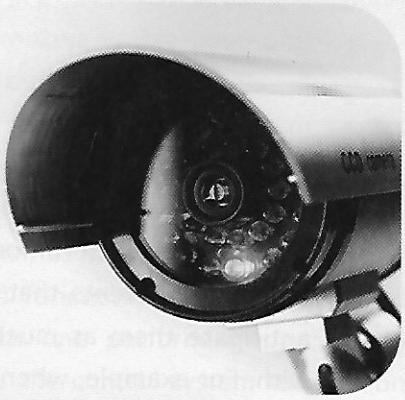


**Image 7.10: A CCTV camera**

[6] Electronic tagging. (26, February 2016). In *Wikipedia, The Free Encyclopedia*. Retrieved 19:30, May 16, 2016, https://en.wikipedia.org/wiki/Electronic_tagging

[7] Prisoner e-tagging a 'resounding success'. (23, May 2012). In iTWeb. Retrieved 19:08, May 28, 2016, http://www.itweb.co.za/index.php?option=com_content&view=article&id=55068

electronic tags could also be placed on immigrants, preventing them from staying in a country illegally.

- **Surveillance, CCTV (Closed Circuit Television)**: Over the past years a large number of cameras have been installed on highways and cities in various countries around the world. CCTV cameras can often be spotted on top of or near traffic lights and intersections, in parks, outside shops or any other position. People are now so accustomed to them that they do not even notice their existence. Recent advances in computer graphics and artificial intelligence have enabled CCTV systems to use algorithms for facial recognition, as well as license plate recognition, to locate or follow a target.

  CCTV supporters claim that by using this surveillance technique, street and shop crimes are reduced, drivers follow the rules of the road, and in general individuals are not tempted to perform illegal acts.

  However, there are a number of questions to be answered:

  - How many CCTV cameras should operate in an area before surveillance becomes excessive, to the point of loss of privacy?
  - Can the information from surveillance techniques ever be misused or fall in the wrong hands?
  - Do surveillance techniques actually reduce crimes or do they just alter it (for example, move it to some other non-surveilled area, or within households, where CCTV cannot be installed yet)?
  - Should workplaces install CCTV cameras or other electronic means to monitor their workers?
  - Should schools install CCTV cameras or other electronic means to monitor both students and teachers?
  - Are citizen rights more important than prevention of potential wrongdoing?

  Surveillance has a myriad of social impacts and ethical considerations that must be taken into account. Most forms of electronic surveillance systems are embedded control systems.

- **Improved safety systems**: After any system, from an elevator to a car, has been developed and deployed into the wider world, it is subject to everyday use. This use may not always be as intended. For example, too many people may enter an elevator, increasing its weight, beyond what can be handled by the elevator's motor. Another example could be a car driven into a wall. Safety systems are put in place so as to avoid such unfortunate events. However, since everyday life is full of unexpected events that cannot be predicted, safety systems improve in order to anticipate them as much as possible. Sometimes these events are properly dealt with. For example, when too many people enter an elevator, a warning light comes on and the elevator does not move until enough people have left. Another example could be the airbags deployed during a car crash to protect the passengers.

In general, safety systems are put into place to protect anything that might be harmed when something does not go as planned. Safety systems are upgraded and improved when they are used but fail for some reason. For example, front airbags were developed to avoid injuries during car crashes. However, in a number of accidents, cars had been struck from the sides. The passengers were injured, since the front airbags did not protect them. Side airbags were created to minimize harm in these accidents. The airbag, as a safety system, was improved. Car brakes are a similar case of the new safety system being improved from the failure of the previous safety system. Forcefully using the car brakes resulted in the locking of the wheels which generated the car skidding, which led to a larger distance covered before the car stopped, compared to when the wheels did not lock. Following these observations, the Anti-lock Braking System (ABS) was developed, to improve the car braking safety system, by preventing the wheels from locking and skidding. This way the stopping distances decreased.



**Image 7.11: Airbag safety system**

## Distributed systems

### 7.1.7 Comparison of centrally controlled systems with distributed systems

| Exit skills. Students should be able to[1]: |
| --- |
| Compare centrally controlled systems with distributed systems. Technical hardware details are not expected. |

In its simplest form, a **centrally controlled** system is a system with a dedicated computer system that is responsible for all the necessary calculations, as well as anything else that needs to be performed so that the control system can function. On the other hand, **distributed** systems have multiple computers, with each one performing part of the necessary calculations, as well as anything else needed for the control system to function. There are arguments to be made for both systems.

- **Centralized systems**: All computer resources reside in a single computer system. Client devices need to be able to connect efficiently and effectively to that single computer system.

The benefits are (usually) lower operational costs (as each client needs minimal hardware), greater security (as all data is centrally stored), less administrative overhead and backup complexity (since all resources are in a single location), as well as greater overall control.

One of the disadvantages is that clients need to connect to the central system to perform any necessary actions, making the connection a probable point of failure. If the single computer system or the connection to it fails, the whole system is not able to function effectively.

- **Distributed systems**: Computer resources are distributed across various systems and each system is self-sustaining for the most part. A central system may exist that coordinates the rest of the systems in some way, but each system has all the necessary resources to perform any necessary action on its own, without needing to communicate with the central system every time. Even if there is no connection to the central system, there would be no catastrophic failure and the distributed system, as a whole, would still function correctly. Distributed systems are easily expandable since self-sustained systems may be added or removed at any point in time without affecting the overall system.

  The disadvantage to this approach is (usually) its cost. It requires additional hardware to be present at each system, so that it can operate on its own, as well as special software that allows each system to do so. Backup of the system requires more bandwidth or resources at each site, adding to the cost and the complexity of the system. Security can also be an issue, since it is more difficult to control and secure a number of systems than a single one.

Cost is a variable factor when comparing centralized to distributed systems. In general, distributed systems require additional hardware that increases the cost of the whole system. However, centralized systems require a single computer system that is very powerful in order to efficiently and effectively coordinate the system as a whole. As such, the cost of a very powerful computer system and its maintenance may be higher than the cost of the additional hardware needed in distributed systems, or vice versa.

Here is an example of traffic signal control systems[8], which may be either centrally-controlled or distributed:

- **Centrally controlled systems**: Every light on each intersection depends on the availability of an effective communications network. Every light needs to be able to communicate with the central computer in a real-time fashion and will not be able to operate, at least as effectively, if the communication is lost. If the central computer stops operating, all the intersections are affected and stop operating effectively. Fault-tolerant systems exist, with two identical central computers, so that if one fails the other can take over. Also, a central computer has a maximum load limit, meaning that it cannot be responsible for real-time coordination of an unlimited number of traffic lights at intersections. A central control system is not

[8] Traffic Signal Control Systems. (2011). In *United States Department of Transportation, Research and Innovative Technology Administration*. Retrieved 18:50, May 20, 2016, http://ntl.bts.gov/lib/jpodocs/edldocs1/13480/ch3.pdf

easily expandable and requires a significant amount of investment in order to do so. However, centralized real-time adaptive control algorithms may easily be installed and real-time surveillance of the system allows for better control of the system.

- **Distributed systems**: A powerful and robust control system at the intersection is responsible for the traffic lights. It does not receive or transmit mandatory real-time control commands over the communications network. It can even operate when the central computer (that is responsible for the grid of traffic signal control systems) is not available. Connection to the central computer is limited and only necessary for synchronization purposes. Expansion of the system is easy, just by adding new traffic lights and their infrastructure, and does not affect the rest of the system. Centralized real-time adaptive control algorithms and surveillance are not available, but may be substituted by local adaptive control algorithms runnig at each intersection.

### 7.1.8 The role of autonomous agents acting within a larger system

| Exit skills. Students should be able to[1]: |
| --- |
| Explain the role of autonomous agents acting within a larger system. Technical hardware details are not expected. |

**Agents** can be anything that can perceive its environment, through sensors, and act upon it, through **effectors**. Examples of agents may be humans (that have eyes, ears and other sensors) to robotic agents (that have cameras, sonars and other sensors) and software agents (that have sensors in the form of bits and can sense their digital environment through those sensors).

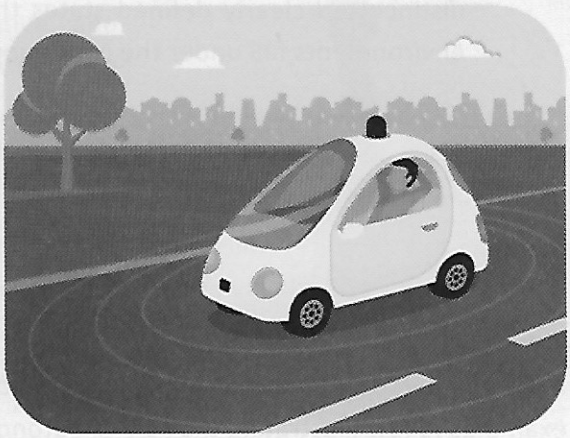


Image 7.12: Autonomous agent (car)

Autonomous agents are entities operating on behalf of an owner (usually a user or another program) with a degree of autonomy and with minimal to no interference from the owner. These agents follow algorithms supplied by the owner to achieve some desired goal(s). Examples of autonomous agents include autonomous robots, software agents that search the world wide web, software computer viruses, etc. Autonomous agents perform actions that depend on their own "experiences" through their sensors. They can apply different sets of pre-programmed actions in different situations and even build their own set of actions through learning algorithms. Autonomous agents display artificial intelligence in that they need to "reason" according to their acquired knowledge. The representation of knowledge is a key-concept for those agents.

The environment within which agents operate plays a significant role in the design, implementation and effectiveness of the agents. Most of the times, agents do not have complete control over their environments. They can perceive and influence some of the environment, and the environment may then influence the agents in return. The complexity of the agent's decision-making is affected by the properties of the environment[9]:

- An environment may be **accessible** or **inaccessible**, and may or may not be able to obtain complete and accurate information of the environment. Most real world environments fall under the inaccessible category, where the environment is too big for an agent to be able to sense it as a whole and senses only a subset.
- An environment may be **deterministic** or **non-deterministic**, according to whether an agent's action will have a pre-defined, guaranteed, effect without any possibility of uncertainty or not. In a deterministic environment, the next state of the environment is determined by its current state and the agent's action. Most real world environments fall under the non-deterministic category, in that they are too complex, are affected by various elements, and probably ever-changing.
- An environment may be **episodic** or **non-episodic**, according to whether an agent's actions are divided into "episodes" or not. If they are divided into "episodes", the next episodes do not depend on the previous episodes and an action's output will not depend on what actions the agent took before. Most real world environments fall under the non-episodic category, in that the agent's earlier actions affect the future actions.
- An environment may be **static** or **dynamic**, and may or may not change while the agent is deliberating an action. Most real world environments fall under the dynamic category, in that an environment is independent of the agent and does not wait for an agent's action before it changes.
- An environment may be **discrete** or **continuous**, depending on the number of distinct and clearly defined states that it may or may not be in. Most real world environments fall under the continuous category.

**Useful Information**: The book "Artificial Intelligence: A Modern Approach"[10] by Stuart Russell and Peter Norvig is a great read and resource that goes well beyond the IB syllabus. Chapter 2, which deals with Intelligent Agents can be read at link: http://www.cs.berkeley.edu/~russell/aima1e/chapter02.pdf

Having understood the meaning of autonomous agents and their environment, a number of examples follow illustrating the role of autonomous agents within a larger system.

[9] Stuart Russell, Peter Norvig. Artificial Intelligence: A Modern Approach, Upper Saddle River (New Jersey, 1995)

- **Autonomous agents in space missions**[10]: Autonomous agents may be of help in both unmanned and manned missions. One type of unmanned mission that needs to take place every day, all day (24/7) is the control and coordination of satellites orbiting Earth. Before the use of autonomous agents, 2-3 persons were needed per satellite. With the increase in the number of satellites this led to a large number of staff responsible for their control, which resulted (amongst other things) in a large financial burden.

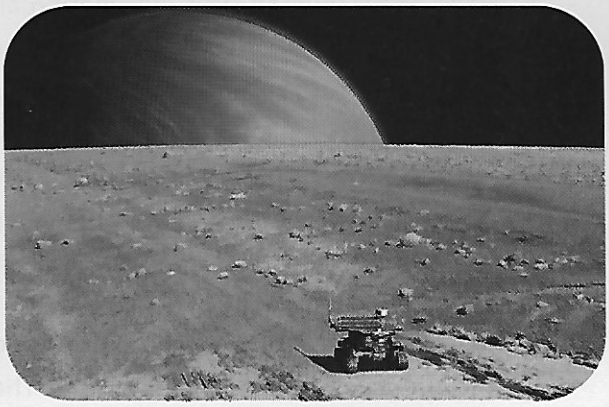


**Image 7.13: Autonomous space agent**

Another type of unmanned missions with difficult and/or restricted time frame communication with the device(s) launched in space, are the deep space missions. Sending a message from Earth to a planet far away, such as Mars or Jupiter, takes minutes or even hours, and may only be available for only a few hours per day. The bandwidth of the connection is also limited. As such, these devices cannot be controlled in a real-time manner. However, they need to be able to act in a real-time manner, as well as be able to face any situations when communication with Earth is unavailable. Autonomous agents allow deep space mission devices to perform real-time tasks when connection to Earth is limited or non-existent, preventing pitfalls and unwanted situations.

In manned missions, the crew spends a long period of time, each day, in monitoring and maintaining. Monitoring is a repetitive and mundane task. Furthermore, most mistakes in manned missions are made by humans that may let something go unnoticed due to negligence or tiredness. Autonomous agents can be responsible for monitoring both everyday tasks, as well as human actions while performing some maintenance tasks.

Of course, it is impossible to take into account all the possible events that could occur. As such, autonomous agents that do not request human intervention in unfamiliar circumstances could lead to errors and to the detriment of a mission.



**Image 7.14: Probable future autonomous agent picker robot**

[10] Stefan Büttcher. Autonomous Agents in Space Missions. Course project for CS886 – Multi-Agent Systems for Real-World Applications, University of Waterloo, Spring 2004. Retrieved 19:25, May 31, 2016, http://stefan.buettcher.org/cs/cs886/project.pdf

- **Autonomous agents in warehouses**[11]: Autonomous agent robots named "Kiva" (or Amazon robots or bots) are already operating in Amazon's warehouses. These robots carry shelves of products to human workers, who in turn pick the items that need to be shipped. This automation allows for the shipping of a far larger number of items to costumers than just using human workers. Apart from these bots, Amazon created the Amazon Picking Challenge (APC), in which researchers competed for building a new bot that would be able to perform the item picking process, instead of human workers. The challenge was hard and most teams failed completely, while the ones that managed to perform better still performed a lot worse than humans. However, the challenge showed that these difficulties might soon be overcome. In all likelihood, bots controlled by autonomous agents will soon be used in warehouses around the world.
- **Autonomous agents in cars**[12, 13, 14]: The idea of fully autonomous cars has been a worldwide endeavor for many years. The DARPA (Defense Advanced Research Projects Agency) Grand Challenge was the first ever competition to formalize this aim, offering prize money to the team able to build a driverless car that could avoid obstacles and reach a certain goal. Since that time, a number of autonomous vehicle prototypes and tests have been conducted, in both urban and rural settings, with increasing success. Autonomous agents are the "brain" of every such autonomous vehicle. Agents use sensors to read information about the car's internal and external states, process the information, and act on it. Agents need to act correctly in an inaccessible, non-deterministic, non-episodic, dynamic and continuous environment that is extremely complex.
- **Autonomous agents in video games**[15]: As video games have improved in quality throughout the years and the industry has grown to be one of the most important in the entertainment business, games have pushed the barriers further in various fields of computer science (including graphics and artificial intelligence), in order to get better, more realistic appearances and behaviors. This need for complex and sophisticated environments cannot depend on pre-programmed elements that loop or always repeat the same patterns of behavior. Agents need to be installed reacting in a manner that will not repeat over time.



**Image 7.15: Autonomous agents in games**

[11] Amazon, robots and the near-future rise of the automated warehouse. (26, January 2016). In *TechRepublic*. Retrieved, 18:00, May 31, 2016, http://www.techrepublic.com/article/amazon-robots-and-the-near-future-rise-of-the-automated-warehouse/

[12] DARPA Grand Challenge. (25, April 2016). In *Wikipedia, The Free Encyclopedia*. Retrieved 19:52, May 31, 2016, https://en.wikipedia.org/wiki/DARPA_Grand_Challenge

[13] Autonomous car. (26, May 2016). In *Wikipedia, The Free Encyclopedia*. Retrieved 19:55, May 31, 2016, https://en.wikipedia.org/wiki/Autonomous_car

[14] Google Self-Driving Car project, https://www.google.com/selfdrivingcar/

[15] Thannia Blanchet. Autonomous Agents in Videogames. Retrieved 19:25, May 31, 2016, http://www.cs.unm.edu/~pdevineni/papers/Blanchet.pdf

## End of chapter example questions with answers

**Exercise 1:** A control system is used to control an elevator car to allow people to go to their desired floor in a building.

1. Identify various types of sensors in this system.
2. Identify other pieces of hardware, other than sensors, that are part of the control system.
3. Outline the sequence of steps that will take place within the computer control system, when a person presses the button to call the elevator.
4. Define the term interrupt, as well as a situation in which it may occur in this system.

**Answer to Exercise 1:**

1. Three sensors that will be present in this system are: proximity, motion and pressure.
2. Other pieces of hardware may include a microprocessor, actuators, transducers, as well as analogue-to-digital and digital-to-analogue converters.
3. 
    a. When a person presses the button, the pressure sensor is activated.
    b. The signal is sent to the microprocessor that determines when the elevator car should arrive at the floor where the person resides.
    c. A signal is sent to the actuators/transducers to move the car to the floor on which the person pressed the button.
    d. After arriving on the floor, another signal is sent to the actuators of the automatic door to open the car doors.
    e. After a fixed time and if there is no other input in the motion sensor of the automatic doors, the doors close.
    f. A signal is sent to the actuators to move the car to the desired floor.
    g. After arriving on the floor, another signal is sent to the actuators of the automatic door to open the car doors so that the person steps out.

    This sequence of steps is a simplified version of what might happen in real life, since a microcontroller must take into account a number of other factors. For example, other people may need to be served at the same time or an indication that the maximum elevator car weight was reached.
4. A signal sent to the processor (by either software or hardware) requesting the processor's immediate attention to a specific event is an interrupt. An example of where this might be used in the elevator control system is the following. After the elevator reaches a desired floor and opens its automatic doors to unload and load people, the doors start closing. At that point a person that wants to enter the elevator puts his/her foot between the two closing doors. A (light or pressure) sensor would sense that the doors did not close and that there is an

object between them and would send an interrupt to the processor to open the doors.

**Exercise 2:** Discuss how computer control systems have been used throughout industries to replace human workers in various accounts.

**Answer to Exercise 2:** Computer control systems can work 7 days a week, 24 hours a day (24/7), all year round. They also work tirelessly and accurately, performing monotonous tasks without any complaints. They can also be used in dangerous environments, such as radioactive, or where there are a lot of fumes, fires, etc., or in environments that cannot be (easily) reached by humans, such as deep sea, space, etc. The initial cost of a computer control system is usually considerable, but once set up, the system will be more economical in the long run. Also, it will reduce labor cost, may increase the quality of work, as well as performance and productivity. Finally, since there will be fewer workers involved, safety will increase. However, such systems may lead to an increase of unemployment and workers made obsolete will need to retrain in order to find a different kind of job.

**Exercise 3:** Describe how a GPS device can locate its position.

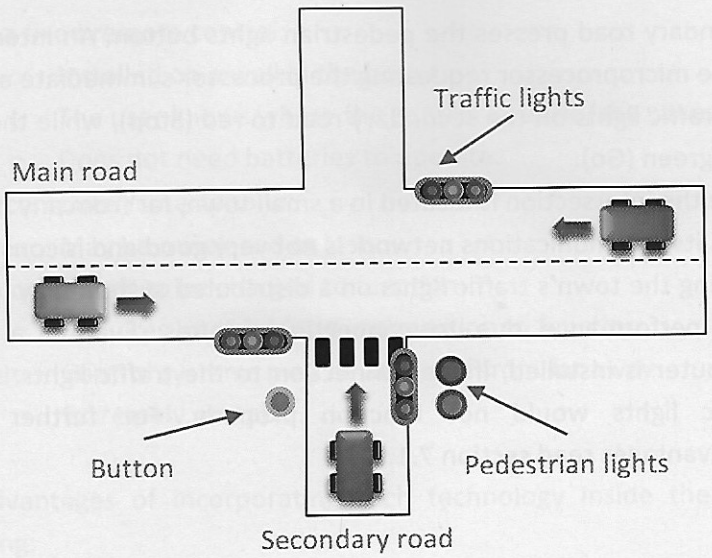
**Answer to Exercise 3:** Read section 7.1.1.

**Exercise 4:** Describe how control traffic lights can function dynamically.

**Answer to Exercise 4:** Read section 7.1.1.

**Exercise 5:** An intersection of a main road and a secondary road is regulated by a set of traffic lights. The secondary road is a one way street. Pedestrian lights are also present allowing people to cross the secondary road.

- By the press of a button, the pedestrian lights, as well as the traffic lights on the main road change to green (Go), while the traffic lights on the secondary road turn red (Stop), allowing people to cross the road.
- If no pedestrians want to cross and there is a vehicle on the secondary road, the pedestrian lights and the traffic lights on the main road turn red (Stop) and green (Go) on the secondary road.
- Otherwise, on all other accounts, every two minutes the traffic lights on the main road, as well as the pedestrian lights, change from green (Go) to red (Stop), while the traffic lights on the secondary road change from red (Stop) to green (Go), and vice versa.




1. Suggest how the sensors and the microprocessor controlling the traffic lights may "sense" a vehicle approaching on the secondary road and perform all the necessary steps.
2. Define the term interrupt, as well as a situation in which it may occur in this system.

The intersection is located in a small town, far from any major cities. As such, its communications network is not very good and is constantly unavailable.

3. The traffic lights installed in the intersection can either belong to a centrally controlled system or a distributed one. Discuss the advantages and disadvantages of both.

**Answer to Exercise 5:**

1.
   a. Either a touch/weight sensor in the road or a camera on top of the traffic lights may be installed to detect an approaching or waiting vehicle on the secondary road.
   b. An analogue-to-digital converter is used to convert the analogue signal from the touch/weight sensor to its digital counterpart. The same applies to the camera.
   c. The digital signal (either the weight or an image) is processed by the microprocessor.
   d. Finally, a signal is sent to the traffic lights so that the pedestrian and the main road traffic lights turn red (Stop), while the traffic lights on the secondary road turn green (Go).
2. A signal sent to the processor (by either software or hardware), requesting the processor's immediate attention to a specific event, is an interrupt. An example of where this might be used in the traffic lights control system is the following:
   A vehicle approaches from the secondary road and the sensors send a signal to the microprocessor. At the same time, a pedestrian that wants to cross the

secondary road presses the pedestrian lights button. An interrupt signal is sent to the microprocessor requesting the processor's immediate attention, changing the traffic lights on the secondary road to red (Stop), while the pedestrian lights turn green (Go).

3. Since the intersection is located in a small town, far from any major cities, and as such, its communications network is not very good and is constantly unavailable, running the town's traffic lights on a distributed system may lead to poor traffic light performance due to connection failures. Even if a powerful central computer is installed, if the connection to the traffic lights is not reliable, the traffic lights would not function properly. For further advantages and disadvantages read section 7.1.7.

**Exercise 6:** Discuss the ethical and social implications of forcing people to carry GPS devices. Provide specific examples.

**Answer to Exercise 6:** Read section 7.1.6.

**Exercise 7:** Discuss the ethical and social implications of countries and/or individuals installing CCTV cameras throughout cities and/or around their homes.

**Answer to Exercise 7:** Read section 7.1.6.

**Exercise 8:** A number of control systems may be used around the house such as:

- smart heating systems that turn on/off automatically as needed
- automatic doors/windows that open/close or even lock as needed
- lighting that turns on/off/dim or even changes colors as needed
- air conditioning that turns on/off as needed
- automatic cleaning that turns on/off as needed

1. Discuss how a user may access these systems.
2. Discuss two advantages of incorporating such technology inside the house.

**Answer to Exercise 8:**

1. The control systems described, as well as any other control system centered around the house could be accessed and controlled in one of the following ways:
   - Via a remote control:
     - May be misplaced within the house.
     - Needs batteries or to be charged to function.
     - Portable enough to access the systems from anywhere in the house.

- Via a fixed remote control:
    - Installed on a wall in the house.
    - The user knows where the control is, as he/she cannot move it.
    - Does not need batteries to operate.
- Via computer/smartphone:
    - May be misplaced or lost.
    - Needs to be charged to function.
    - Requires internet/Wi-Fi connection to function.
    - Control systems may be accessed from far away.
    - Extremely portable.

2. The advantages of incorporating such technology inside the house include the following:
    a. Cost reduction: By scheduling exactly when these control systems function, one can control expenses, reduce costs and save energy.
    b. Improved comfort: By scheduling exactly when these control systems function, one can cater for his/her specific needs. For example, he/she can turn the heater on just before leaving work and arrive to a warm house.

## Chapter References


1. International Baccalaureate Organization. (2012). IBDP Computer Science Guide.
2. Control System. (12, May 2016). In Wikipedia, The Free Encyclopedia. Retrieved 19:05, May 13, 2016, from https://en.wikipedia.org/wiki/Control_system
3. Smart thermostat. (19, April 2016). In Wikipedia, The Free Encyclopedia. Retrieved 19:25, May 13, 2016, https://en.wikipedia.org/wiki/Smart_thermostat
4. Dimitriou K. Hatzitaskos M. (2015). Core Computer Science for the IB Diploma Program. Athens: Express Publishing. More information at: https://www.expresspublishing.co.uk/gr/en/content/core-computer-science-ib-diploma-program
5. List of sensors. (19, April 2016). In Wikipedia, The Free Encyclopedia. Retrieved 19:30, May 13, 2016, https://en.wikipedia.org/wiki/List_of_sensors
6. Electronic tagging. (26, February 2016). In Wikipedia, The Free Encyclopedia. Retrieved 19:30, May 16, 2016, https://en.wikipedia.org/wiki/Electronic_tagging
7. Prisoner e-tagging a 'resounding success'. (23, May 2012). In iTWeb. Retrieved 19:08, May 28, 2016, http://www.itweb.co.za/index.php?option=com_content&view=article&id=55068
8. Traffic Signal Control Systems. (2011). In United States Department of Transportation, Research and Innovative Technology Administration. Retrieved 18:50, May 20, 2016, http://ntl.bts.gov/lib/jpodocs/edldocs1/13480/ch3.pdf
9. Stuart Russell, Peter Norvig. Artificial Intelligence: A Modern Approach, Upper Saddle River (New Jersey, 1995)
10. Stefan Büttcher. Autonomous Agents in Space Missions. Course project for CS886 – Multi-Agent Systems for Real-World Applications, University of Waterloo, Spring 2004. Retrieved 19:25, May 31, 2016, http://stefan.buettcher.org/cs/cs886/project.pdf
11. Amazon, robots and the near-future rise of the automated warehouse. (26, January 2016). In TechRepublic. Retrieved, 18:00, May 31, 2016, http://www.techrepublic.com/article/amazon-robots-and-the-near-future-rise-of-the-automated-warehouse/
12. DARPA Grand Challenge. (25, April 2016). In Wikipedia, The Free Encyclopedia. Retrieved 19:52, May 31, 2016, https://en.wikipedia.org/wiki/DARPA_Grand_Challenge
13. Autonomous car. (26, May 2016). In Wikipedia, The Free Encyclopedia. Retrieved 19:55, May 31, 2016, https://en.wikipedia.org/wiki/Autonomous_car
14. Google Self-Driving Car project, https://www.google.com/selfdrivingcar/
15. Thannia Blanchet. Autonomous Agents in Videogames. Retrieved 19:25, May 31, 2016, http://www.cs.unm.edu/~pdevineni/papers/Blanchet.pdf

# TOPIC D – Object-oriented programming

## Topic D – Object-oriented programming[1]

### Tool used

All Java programs were written and tested in BlueJ, which can be downloaded from the following link:

**http://www.blueJ.org/**

The BlueJ development environment was created by the University of Kent and is ideally suited for students.

**Useful Information**: Although it is assumed that all students who choose to take Topic D – Object Oriented Programming have mastered (up to a point) coding in Java, there is no easy or slow introduction in the syllabus content.
As such, it is up to the students or the teachers to find books, online material or presentations on programming with Java. This is not always easy to do and although there is a plethora of tutorials on the Internet, some may be very advanced, some may not be complete and some may not even be accurate. As such, a list of helpful resources are given below. These free resources are most helpful and should be at least considered by any student learning Java. They are presented in the order in which they should be studied. They also include exercises.

1. **Introduction to Java Programming:**
   http://www.ntu.edu.sg/home/ehchua/programming/java/J1a_Introduction.html
2. **Java Programming Tutorial:**
   http://www.ntu.edu.sg/home/ehchua/programming/java/J2_Basics.html
3. **Java Exercises:**
   https://www3.ntu.edu.sg/home/ehchua/programming/java/J2a_BasicsExercises.html
4. **Java – Tutorial:**
   http://www.tutorialspoint.com/java/index.htm

[1] International Baccalaureate Organization. (2012). IBDP Computer Science Guide.

# D.4 Advanced program development

## D.4.1 The term "recursion"

| Exit skills. Students should be able to[1]: |
| --- |
| Define the term recursion. |

Recursion was identified and explained in sections 5.1.1 to 5.1.3, "Thinking recursively". Situations where recursion is useful were presented and recursive algorithms were traced.

Recursion is a specific approach to problem solving in computer science. In this approach, the solution to a problem depends on solutions to smaller instances of the same problem.[2]

In programming, recursion happens when a procedure (method) calls itself until some terminating condition is met. This is accomplished without any specific repetition construct, such as a `while` or a `for` loop. Recursion follows one of the basic problem solving techniques, which is to break the problem at hand into smaller subtasks (divide and conquer). Most algorithms that may be presented in a recursive manner may also be presented in an iterative manner and vice versa.

## D.4.2 Application of recursive algorithms

| Exit skills. Students should be able to[1]: |
| --- |
| Understand that recursion can be applied to a small subset of programming problems to produce elegant solutions.<br>Understand that recursive algorithms are rarely used in practice.<br>LINK Think abstractly, think recursively. |

In some cases, recursive algorithms can produce "simpler" and "cleaner" code compared to their iterative counterparts if used wisely. This code is easier to understand and maintain and, as such, useful to have, when possible. Sometimes "simpler" and "cleaner" code is preferred even if it is not the most efficient. Below, a number of algorithmic examples are compared, solved in both iterative and recursive versions:

- **Factorial**: A factorial function is defined mathematically as follows:

$$n! = \begin{cases} 1, if\ n = 0 \\ n * (n-1)!, if\ n > 0 \end{cases}$$

  4! = 4*3*2*1*1 = 24. The following two code snippets find the factorial of a positive integer number `n`. The left-hand is an iterative version, while the right-hand is a recursive version.

[2] Recursion (computer science). (27, May 2016). In *Wikipedia, The Free Encyclopedia*. Retrieved 13:39, June 3; 2016, https://en.wikipedia.org/wiki/Recursion_(computer_science)

| Iterative version | Recursive version |
|---|---|
| `public int fact(int n) {`<br>`  int answer = 1;`<br>`  for (int i = 2; i <= n; i++)`<br>`  {`<br>`  answer = answer * i;`<br>`  }`<br>`  return answer;`<br>`}` | `public int fact(int n) {`<br>`  if (n == 0) return 1;`<br>`  return n * fact(n - 1);`<br>`}` |

Both of these versions are "simple", "clean" and can be understood easily enough. The iterative version is a little longer and uses a **`while`** loop, while the recursive version includes a base case (when **`n`** is equal to **`0`**) and calls itself **`n`** times. Each time the recursive version calls itself, it causes some memory to be allocated.

Figure D.1 presents how memory is allocated every time a recursive function calls itself, assuming that the factorial of number 4 is to be found (so **`fact(4)`** is called). After **`fact(4)`** needs to return **`4 * fact(3)`**. To do that, **`fact(3)`** is called, while **`fact(4)`** is still running (it has not returned yet, since it has called **`fact(3)`**). **`fact(3)`** needs to return **`3 * fact(2)`** and so **`fact(2)`** is called (with **`fact(4)`** and **`fact(3)`** still running). **`fact(2)`** needs to return **`2 * fact(1)`**. So, **`fact(1)`** is called that needs to return **`1 * fact(0)`**. **`n == 0`** is the base case and so **`fact(0)`** simply returns **`1`**. The base case is the case that stops the recursion from

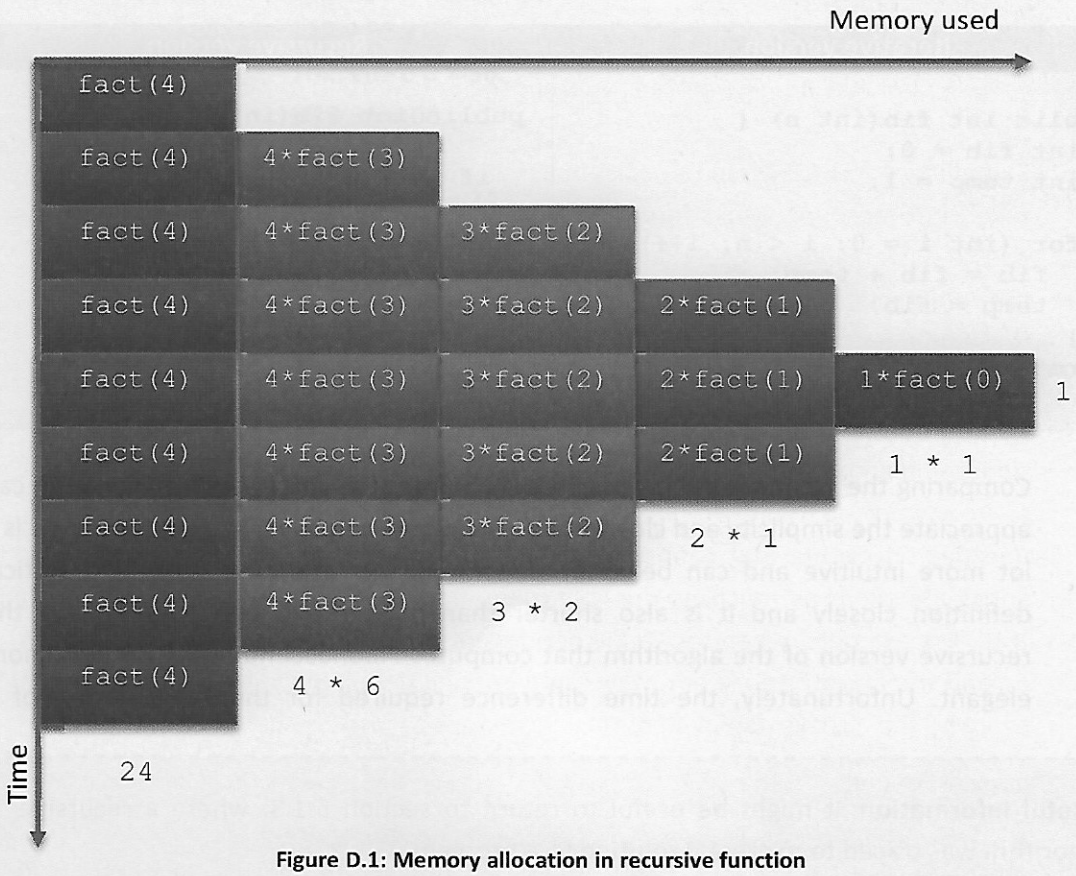



**Figure D.1: Memory allocation in recursive function**

running infinitely. It does not call another `fact` method. At this point, `fact(4)`, `fact(3)`, `fact(2)`, `fact(1)` and `fact(0)` are all taking up space in the memory. Finally, `fact(0)` first returns, so `fact(1)` may then return, so `fact(2)` may return, so `fact(4)` may return at the end with the final result. The recursive version of the factorial algorithm to compute the factorial of `n` generates `n-1` recursive calls, rendering this algorithm inefficient compared to its iterative implementation.

As it is becoming apparent, calling methods recursively uses up memory quickly and can result in the system stack running out of memory (as explained in sections 5.1.6-5.1.7). In this example, this could happen if a large number were given to the recursive version of the factorial algorithm. Recursion is a powerful concept and tool that allows for elegant, simple solutions to problems, but may result in inefficient algorithms if not used wisely.

- **Fibonacci**: A Fibonacci function is defined mathematically as follows:

$$F_n = \begin{cases} 0, if\ n = 0 \\ 1, if\ n = 1 \\ F_{n-1} + F_{n-2}, if\ n > 1 \end{cases}$$

Every number after 2 is the sum of the preceding two numbers. So, the first Fibonacci numbers are the following: 1, 1, 2, 3, 5, 8, 13, 21 ...

The base cases of a recursive algorithm would be $F_0$ and $F_1$.

| Iterative version | Recursive version |
|---|---|
| `public int fib(int n) {`<br>`  int fib = 0;`<br>`  int temp = 1;`<br><br>`  for (int i = 0; i < n; i++) {`<br>`  fib = fib + temp;`<br>`  temp = fib;`<br>`  }`<br>`  return fib;`<br>`}` | `public int fib(int n) {`<br><br>`  if (n < 2) return n;`<br><br>`  return fib(n-1) + fib(n-2);`<br>`}` |

Comparing the iterative to the recursive version for the Fibonacci numbers, one can appreciate the simplicity and cleanness of the recursive code. The recursive code is a lot more intuitive and can be easily understood. It resembles the mathematical definition closely and it is also shorter than its iterative version. As such, the recursive version of the algorithm that computes Fibonacci numbers is much more elegant. Unfortunately, the time difference required for the computation of a

**Useful Information**: It might be useful to return to section 5.1.3, where a recursive algorithm was traced to express a solution to a problem.

Fibonacci number between these two versions is enormous, with the iterative version being a lot faster.

- **Euclidean greatest common divisor (GCD)**: A Euclidean GCD function is defined mathematically as follows:

$$For\ x, y > 0, gcd(x, y) = \begin{cases} x, if\ x = y \\ gcd(x, y - x), if\ x < y \\ gcd(x - y, y), if\ x > y \end{cases}$$

It is probably the oldest recorded recursive algorithm in history (around 300 B.C.). To find the greatest common divisor of two numbers, x and y:

  - Subtract the smaller of the two numbers from the larger one repeatedly, until the resulting difference d is smaller than the smaller of the two initial numbers, x and y.
  - Repeat the same steps with d and the smallest of the two numbers from the previous step.
  - Continue until the two numbers are equal. That number is the GCD of the two initial number.

For example, if initial x is 420 and y is 96 then the algorithm would work as follows:

**Step 1:** 420 – 96 = 324,
324 – 96 = 228,
228 – 96 = 132,
132 – 96 = 36.
**Therefore d = 36.**

**Step 2:** 96 – 36 = 60,
60 – 36 = 24.
**Therefore d = 24.**

**Step 3:** 36 – 24 = 12.
**Therefore d = 12.**

**Step 4:** 24 – 12 = 12.
**Therefore d = 12 and both the smallest of the two numbers of this step (24 and 12) and d are 12.**

As such, 12 is the GCD of 420 and 96.

| Iterative version | Recursive version |
|---|---|
| `public int gcd(int m, int n) {`<br>`  while (m != n) {`<br>`  if (m > n) {`<br>`  m = m - n;`<br>`  } else {`<br>`  n = n - m;`<br>`  }`<br>`  }`<br>`  return m;`<br>`}` | `public int gcd(int m, int n) {`<br>`  if (m == n) {`<br>`  return n;`<br>`  } else if (m < n) {`<br>`  return gcd(m, n-m);`<br>`  } else {  //m > n.`<br>`  return gcd(m-n, n);`<br>`  }`<br>`}` |

Both of these versions are simple, clean and can be understood easily enough. However, the recursive version follows the mathematical definition and as such it is probably easier to think of and implement in the first place. On the other hand, the recursive version does call itself multiple times, taking up more memory space.

- **Towers of Hanoi**: The Towers of Hanoi is a puzzle in which the elegance of recursion shines through, compared to an iterative method.
  This classic game consists of three vertical pegs attached to a board, as depicted in Image D.1. The pegs are usually labelled in some way. (ex. A, B, and C). A number of `n` disks that have a hole in their centers, so that they can slide onto the pegs, are mounted on peg A, from largest (at the bottom) to smallest (at the top). The objective of the game is to move all disks from peg A to peg C, one at a time, without having a larger disk on top of a smaller one at any point in time. Image D.1 presents the initial configuration of the game with 7 disks on peg A, a middle state in which the disks are placed in various pegs, and the final configuration of the game, the winning state, with 7 disks on peg C.

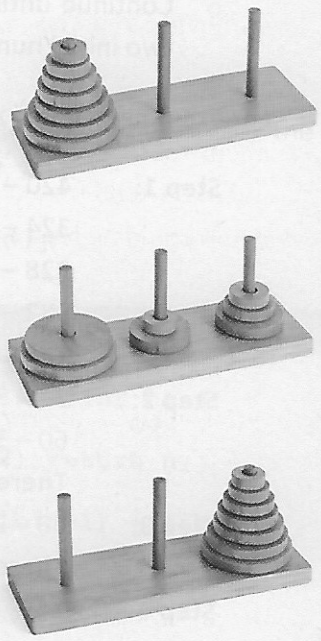


**Image D.1: The Towers of Hanoi**

  This seemingly complicated problem becomes easy when seen through the lens of a recursive approach. There are only three steps that need to be performed in order to solve it:

  **Step 1:** Move the smaller n-1 disks from peg A to peg B.
  **Step 2:** Move the remaining disk from peg A to peg C.
  **Step 3:** Move the smaller n-1 disks from peg B to peg C.

  The solutions to the Towers of Hanoi problem for one (n=1), two (n=2) and three disks (n=3), using the above algorithm, are outlined below:

- **For one disk, where n=1:**
  1. Move the one disk from peg A to peg C *[Step 2]*

  (steps 2 and 4 do nothing since n-1, for n=1, is equal to 0).

- **For two disks, where n=2:**
  1. Move the top disk from peg A to peg B. *[Step 1]*
  2. Move the remaining disk from peg A to peg C. *[Step 2]*
  3. Move the top disk from peg B to peg C. *[Step 3]*

- **For three disks, where n=3:**
  1. Move the top disk from peg A to peg C.
  2. Move the second disk from peg A to peg B. *[Step 1]*
  3. Move the top disk from peg C to peg B.
  4. Move the remaining disk from peg A to peg C. *[Step 2]*
  5. Move the top disk from peg B to peg A.
  6. Move the second disk from peg B to peg C. *[Step 3]*
  7. Move the top disk from peg A to peg C.

  Step 1 of the algorithm is made up from moves 1 to 3, step 2 from move 4 and step 3 from moves 5 to 7.

In order to implement the solution to the Towers of Hanoi problem using recursion, a method would be needed that would move **n** disks from one peg to another. That method could be called **moveDisks** and would include the following parameters:

```
moveDisks(int n, char from, char to, char excl)
```

- **n** would represent the number of disks to be moved.
- **from** would represent the peg from which the disks would move
- **to** would represent the peg to which the disks would move
- **excl** would represent the peg that would not be used for the move

When n would be equal to 1, then the single disk would be moved to its designated peg and **moveDisks** would stop. This would be the base case of the recursion.

The three steps described above that would solve the Towers of Hanoi problem, in a recursive manner, are rewritten using the **moveDisks** method described above, as follows:

**Step 1:** Move the smaller n-1 disks using `moveDisks(n-1, A, B, C)`.
**Step 2:** Move the remaining disk using `moveDisks(1, A, C, B)`.
**Step 3:** Move the smaller n-1 disks using `moveDisks(n-1, B, C, A)`.

Taking into account these three steps, the Towers of Hanoi problem could be implemented in the following manner:

```
public static void moveDisks(int n,char from,char to,char excl)
{
  if(n == 1) {
```

```
            System.out.println("Move disk from " + from + " to " + to);
        } else {
            moveDisks(n-1, from, excl, to);
            moveDisks(1, from, to, excl);
            moveDisks(n-1, excl, to, from);
        }
    }
```

An example output of the above code for three disks that are to be moved from peg A to peg C would be the following:

```
1. Move disk from A to C.
2. Move disk from A to B.        [Step 1]
3. Move disk from C to B.
4. Move disk from A to C.        [Step 2]
5. Move disk from B to A.
6. Move disk from B to C.        [Step 3]
7. Move disk from A to C.
```

An iterative version of the Towers of Hanoi algorithm is much more complicated than the elegant, simple and clear solution presented above. Usually, if statements, loops, arrays and/or queues are needed for such an implementation[3].

Through the previous examples one can understand that recursion can be applied to a number of programming problems to produce simple and elegant solutions. However, because recursive algorithms are usually very inefficient, when compared to their iterative counterparts, it is important to note that recursive algorithms are rarely used in practice. In real life computer systems, that have a fixed amount of resources, the algorithms used should be as efficient as possible in order for the system as a whole to be as efficient as possible. In real life, the elegance, "cleanness" and "simplicity" of an algorithm usually come second to efficiency.

### D.4.3 Construction of algorithms that use recursion

| Exit skills. Students should be able to[1]: |
| --- |
| This is limited to a method that returns no more than one result and contains either one or two recursive calls.<br>LINK Connecting computational thinking and program design. |

In section D.4.2 a number of recursive algorithms were described as well as traced. In this section, a simple recursive algorithm is constructed that can be used for the evaluation of exponents of a number (i.e. raising a base number to a power).

Raising a base number **a** to the power of **n** is defined mathematically as follows:

---

[3] Peter Smith. An iterative solution to Towers of Hanoi. Course Comp 151, California State University Northridge, September 2002. Retrieved 22:20, June 5, 2016, http://www.csun.edu/~psmith/151handouts/hanoi3out.pdf

$$a^n = \underbrace{a * \ldots * a}_{n}$$

The base number **a** is multiplied **n** times by itself.
The following example depicts 3 raised to the powers of 0, 1, 2, 3 and 4 respectively:

- $3^0 = 1$
- $3^1 = 3 = 3$
- $3^2 = 3 * 3 = 9$
- $3^3 = 3 * 3 * 3 = 27$
- $3^4 = 3 * 3 * 3 * 3 = 81$

A recursive relation may be deduced and defined as: $a^n = a*a^{n-1}$. The base case for this recursive relation is when $a^0$, which always equals to 1.
The example of 3 raised to the powers of 0, 1, 2, 3 and 4 respectively, described through the lens of the deduced recursive relation $a^n = a*a^{n-1}$, is presented below:

- $3^0 = 1$
- $3^1 = 3 * 3^0 = 3 * 1 = 3$
- $3^2 = 3 * 3^1 = 3 * 3 * 3^0 = 3 * 3 * 1 = 9$
- $3^3 = 3 * 3^2 = 3 * 3 * 3^1 = 3 * 3 * 3 * 3^0 = 3 * 3 * 3 * 1 = 27$
- $3^4 = 3 * 3^3 = 3 * 3 * 3^2 = 3 * 3 * 3 * 3^1 = 3 * 3 * 3 * 3 * 3^0 = 3 * 3 * 3 * 3 * 1 = 81$

From the examples, the recursive relation deduced, as well as the base case, the recursive algorithm could be written in Java in the following manner:

```
public static int power(int a, int n)
{
  //Base case
  if(n == 0) {
    return 1;

  } else {

    //Recursive step
    return a * power(a, n-1);
  }
}
```

In any recursive algorithm there is always a base case. Otherwise the algorithm would not stop and would continue forever or until the resources of the computer system running the algorithm would deplete. As such, there usually exists an **if...else** statement, that determines whether the base case or the recursive step needs to run. It is very important that the base case is always checked first (in this example, the **if(n == 0)** clause is run before the **else** clause).

### D.4.4 Trace of recursive algorithms

| Exit skills. Students should be able to[1]: |
| --- |
| All steps and calls must be shown clearly.<br>LINK Connecting computational thinking and program design. |

In section D.4.2 a number of recursive algorithms were described as well as traced. In this section the simple recursive algorithm that was constructed in section D.4.3, which can be used for the evaluation of exponents of a number (i.e. raising a base number to a power), is traced for when base 3 is raised to the power of 2. The Java method **power** that was implemented in section D.4.3 is used (i.e. **power(3, 2)** is called). The trace table is presented below:

| step | power | n == 0 | return | result |
|---|---|---|---|---|
| 1 | power(3, 2) | false | 3 * power(3, 1) | - |
| 2 | power(3, 1) | false | 3 * power(3, 0) | - |
| 3 | power(3, 0) | true | 1 | 1 |
| 4 | power(3, 1) | - | 3 * 1 | 3 |
| 5 | power(3, 2) | - | 3 * 3 | 9 |

**Table D.1: Trace table for the power(3, 2) method**

There are five steps in the above trace table. **power(3, 2)** is called, **n** is not equal to zero, and **3 * power(3, 1)** is returned. However, before it can return, **power(3, 1)** needs to be called. So, **power(3, 1)** is called, **n** is not equal to zero, and **3 * power(3, 0)** is returned. However, again, before it can return, **power(3, 0)** needs to be called. **power(3, 0)** is called and this time **n** is equal to zero, so **1** is returned. **power(3, 1)** returns **3 * power(3, 0) = 3 * 1 = 3**. Finally, **power(3, 2)** returns **3 * power(3, 1) = 3 * 3 = 9**.

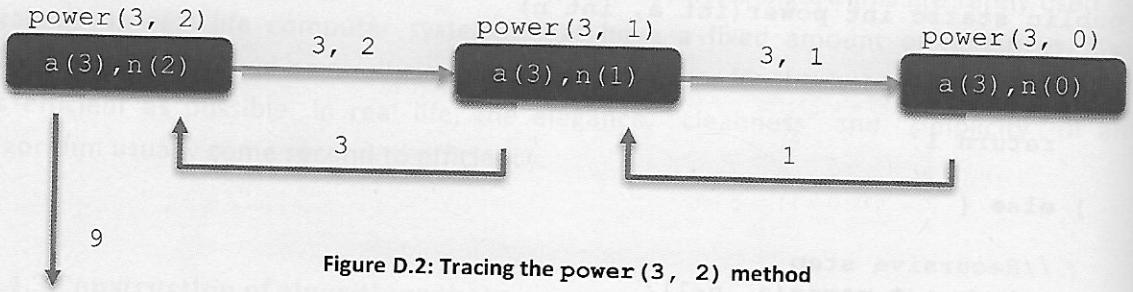



**Figure D.2: Tracing the power(3, 2) method**

It is very important to understand that a recursive method will keep on calling itself, and consuming more system memory, until the base case is reached. At that point the last method called will return, and only then will the second-to-last method return and so on until the initial method can return the final result. When the base case is reached and the methods called return, one by one, in the reverse order they were called, the memory allocated to those methods will be deallocated and freed.

### D.4.5 Define the term object reference

| Exit skills. Students should be able to[1]: |
|---|
| Define the term object reference, as typified by simple classes that are self-referential. |

Reference signifies a value that points to or refers to some data. That data is present either in the computer's memory or its data storage, and may be accessed through the reference. For example, take the following simple program into consideration:

```
public class HelloWorld5Times {
  public static void main(String[] args) {
    int times = 5;

    for(int i = 0; i < times; i++) {
      System.out.println("Hello World!");
    }
  }
}
```

A variable named **times** exists that determines how many times the **for loop** is going to run and how many times the **"Hello World!"** message is going to appear as output. **times** is a name that refers to the number 5 in this example. Since **times** is an **int**, it is a primitive data type. That means that the variable references a memory location and in that location the value is stored. So, 5 is stored in the memory location that is referenced by **times**.

In a similar manner, **object reference** signifies a value that refers to some memory location that holds another memory location where an object's details are stored. For example, take into consideration the following simple object, which describes a (book or magazine) publisher:

```
public class Publisher
{
  //Instance variables
  private String name, address, website;
  private int telephone;

  //Constructor
  public Publisher(String name, String address,
                   int telephone, String website)
  {
   setName(name);
   setAddress(address);
   setTelephone(telephone);
   setWebsite(website);
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getAddress() {
    return address;
  }

  public void setAddress(String address) {
    this.address = address;
  }
```

```
  public String getWebsite() {
    return website;
  }

  public void setWebsite(String website) {
    this.website = website;
  }

  public int getTelephone() {
    return telephone;
  }

  public void setTelephone(int telephone) {
    this.telephone = telephone;
  }
}
```

The **Publisher** object includes four **private** properties/instance variables (three of type **String** and one of type **int**). These instance variables cannot be accessed directly but can be accessed indirectly through the use of the get/set methods. Every **Publisher** is defined as having a name, address, telephone number and a website. The constructor method **Publisher(String name, String address, int telephone, String website)** therefore takes in four parameters. These parameters populate the instance variables of the current object using the set methods. These set methods (ex. **setName(String name)**) make use of the **this** keyword. The **this** keyword works as a reference to the current object instantiation and therefore the current object's instantiation variables. For example, if there are two **Publisher** objects **p1** and **p2**, **p1** has its own instance variables, unaffected by **p2**'s instance variables.

The **Publisher** object is used by the following simple program to create a **Publisher** and output its details. **p** is a variable that refers to the publisher object created. Every object of this type includes some instance variables that may be accessed using the **object.propertyName** rule if they are **public**. For example, the publisher's name of the **publisher** object referenced by **p** could have been accessed through **p.name** if the property **name** was public. However, most object properties, as is the case in our example, are **private**. As such, they can only be accessed through the get/set methods. This approach is safer as it allows various checks to be completed before a property is read or set. For example, when setting a telephone number there may be some rules that need to be followed, such as that telephone numbers cannot be negative numbers. These rules may be placed inside the setter method of the telephone variable (i.e. **setTelephone(int telephone)**). Accessing or setting a variable directly may lead to undesired results.

```
public class ObjectReferenceExample {
  public static void main(String[] args) {
    Publisher p = new Publisher("Express Publishing",
                                "Gardenias 25, Aharnes, Menidi",
                                00302102120800,
                           "https://www.expresspublishing.co.uk");
```

```
        System.out.println("Publisher " + p.getName() + " with
        address " + p.getAddress() + ", telephone number " +
        p.getTelephone() + " and website " + p.getWebsite());
    }
}
```

It is important to understand that a reference is distinct from the data itself. It is only a name used to refer to the data.

### D.4.6 Construct algorithms that use reference mechanisms

| Exit skills. Students should be able to[1]: |
| --- |
| Construct algorithms that use reference mechanisms. |

Expanding on the **Publisher** example described in the previous section, D.4.5, a new **ObjectReferenceExample** class is defined below, which creates two **Publisher** variables **p1** and **p2** that refer to two different **Publisher** objects. Each **Publisher** object has its own instance properties.

```
public class ObjectReferenceExample {
 public static void main(String[] args) {
   Publisher p1 = new Publisher("Express Publishing",
                                "Gardenias 25, Aharnes, Menidi",
                                00302102120800,
                            "https://www.expresspublishing.co.uk");

   Publisher p2 = new Publisher("Random Publishing",
                                "Random St. 42, Random Location",
                                0015100000000,
                            "https://www.randomPublishing.com");


    System.out.println("Publisher " + p1.getName() + " with
    address " + p1.getAddress() + ", telephone number " +
    p1.getTelephone() + " and website " + p1.getWebsite());

    System.out.println("Publisher " + p2.getName() + " with
    address " + p2.getAddress() + ", telephone number " +
    p2.getTelephone() + " and website " + p2.getWebsite());
  }
}
```

The output of the above program would be:

```
Publisher Express Publishing with address Gardenias 25, Aharnes,
Menidi, telephone number 00302102120800 and website
https://www.expresspublishing.co.uk
Publisher Random Publishing with address Random St. 42, Random
Location, telephone number 0015100000000 and website
https://www.randomPublishing.com
```

### Searching and sorting algorithms

This section presents algorithms that may use reference mechanisms, such as linear search, binary search, bubble sort and selection sort. An array named **students** of **Student** objects is taken into consideration. The **Student** with grade **85** is found, with the use of the searching algorithms, while the array is sorted in descending **grade** order, with the use of the sorting algorithms. Each **Student** object has two instance variables (**name**, which is a **String** that holds the name of the student, and **grade**, which is an **int** that represents the student's grade). For simplicity, it is assumed that no two **Student** objects may have the same **grade** in this particular **students** array. The **Student** object is defined as follows:

```
public class Student
{
  //Instance variables
  private String name;
  private int grade;

  //Constructor
  public Student(String name, int grade)
  {
    setName(name);
    setGrade(grade);
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getGrade() {
    return grade;
  }

  public void setGrade(int grade) {
    this.grade = grade;
  }
}
```

- **Linear search**: Searches every element of the **student** array. If the search key matches an element, its index is returned. Otherwise -1 is returned. Since linear search is used to find the student with score 85, the search key in this example is the grade of the student.

```
public int sequentialSearch(Student student, int[] students)
{
  boolean found = false;
  int place = 0;

  while (place < students.length && !found)
  {
    if (student.getGrade() == students[place].getGrade()) {
      found = true;
```

```
    } else {
      place = place + 1;
    }
  }

  if (found) {
    return place;
  } else {
    return -1;
  }
}
```

A similar algorithm was presented in Topic 4.2.1 – Searching, sorting and other algorithms on arrays in the Core Computer Science for the IB Diploma Program book[4]. However, whilst in that version of the searching algorithm only integers (or other primitive data types) were compared, in the algorithm presented above objects are compared. Objects cannot be compared in the same manner as primitive data types. That is, one cannot test for equality between two objects by simply using the equality operator (==). So, in order to determine if the student in the array is the one that is being searched for (i.e. has a score of `85`), the `grade` instance variable of the `student` object is compared to the `grade` of the `student` at the `place` position in the array. The equality operation is displayed in red letters.

- **Binary search**: Searches a sorted version of the `students` array for the student with score `85`. For binary search to work, the array under examination needs to be sorted. In this example, the array is sorted in descending `grade` order (i.e. the student with the best grade is first, whilst the student with the worst grade is last). The binary search algorithm checks the middle element in the array. If that element matches the search key, it is returned and the algorithm stops. Otherwise, the algorithm determines if the element to be found could be located "to the right" or "to the left" of the middle element and proceeds to repeat the aforementioned algorithm for that portion of the array, disregarding the rest of the elements.

```
public int binarySearch(Student student, int[] students)
{
  boolean found = false;

  int low = 0;
  int high = nums.length -1;
  int middle = 0;

  while (high >= low && !found) {
    middle = (low + high) / 2;

    if (student.getGrade() > students[middle].getGrade()) {
      high = middle - 1;
    } else if (student.getGrade() < students[middle].getGrade()) {
      low = middle + 1;
    } else {
```

[4] Dimitriou K. Hatzitaskos M. (2015). Core Computer Science for the IB Diploma Program. Athens: Express Publishing. More information at: https://www.expresspublishing.co.uk/gr/en/content/core-computer-science-ib-diploma-program

```
        found = true;
      }
    }

    if (found) {
      return middle;
    } else {
      return -1;
    }
  }
```

Again, a similar algorithm was presented in Topic 4.2.1 – Searching, sorting and other algorithms on arrays in the Core Computer Science for the IB Diploma Program book[4]. Since objects cannot be compared in the same manner as primitive data types, the **grade** instance variable of the **student** object is compared to the **grade** of the **student** at the **middle** position in the array. This comparison is displayed in red letters.

- **Bubble sort:** Compares every element of the **student** array to its next. If they are not in the correct order, they are swapped. This process is called "a pass". The algorithm continues until an entire pass is completed without any swaps taking place. As such, all elements are in the correct order.

```
public void bubbleSort(int[] students)
{
  Student temp;
  boolean done;

  do {
    done = true ;

    for(int current=0; current<students.length-1; current=current+1) {
      if(students[current].getGrade()<students[current+1].getGrade()) {
        temp = students[current];
        students[current] = students[current+1];
        students[current+1] = temp;
        done = false;
      }
    }
  } while (!done);
}
```

- **Selection sort:** Finds the student with the best grade and swaps him/her with the student at the first position of the array. The student with the second best grade is then found and is swapped with the student at the second position of the array, and so on, until there are no more students to be sorted.

```
public void selectionSort(int[] students)
{
  int first, least;
  Student temp;
```

```
  for(first = 0; first < size; first = first + 1) {
    least = first;
    for(int current=first+1; current<students.length; current++) {
      if(students[current].getGrade()>students[least].grade) {
        least = current;
      }
    }
    temp = students[least];
    students[least] = students[first];
    students[first] = temp;
  }
}
```

### D.4.7 Identify the features of the Abstract Data Type (ADT) list

| Exit skills. Students should be able to[1]: |
| --- |
| Understand the nature of an ADT – where no implementation details are known but the actions/methods are standard. |

*Abstract Data Types (ADTs)* include the phrase *data type* that has been encountered before. For example, Java build-in primitive data types, such as `int`, have already been discussed. The `int` data type includes whole-number values between –2,147,483,648 and 2,147,483,647, as well as arithmetic operators +, -, *, /, etc. These arithmetic operators are an essential part of the `int` data type, as they describe what operations can be performed on it.

Through the use of object-orientated programming one may create classes that represent custom data types. For example, a class that represents time, which includes fields for hours, minutes and seconds, can be considered a custom data type. This class may be added or subtracted just like the `int` data type. However, instead of using operators such as + and -, one would use methods like `add()` and `subtract()`.

Data types may be considered as any class that consists of data (fields/properties) and operations on that data (methods/functions).

**Useful Information**: It is important to understand how objects can be compared. One cannot test for equality between two objects by simply using the equality operator (==). Take into consideration the following object:

```
public class Person
{
  public int idNumber;
  public String name;
  public int age;

  public Student(int idNumber, String name, int age)
  {
    this.idNumber = idNumber;
    this.name = name;
    this.age = age;
  }
}
```

In the following program, two **Person** objects are created and compared as follows:

```
public class ObjectEqualityExample {
  public static void main(String[] args) {
    Person p1 = new Person(318, "Kostas", 18);
    Person p2 = new Person(318, "Kostas", 18);

    if(p1 == p2) {
      System.out.println("p1 is the same as p2");
    } else {
      System.out.println("p1 is not the same as p2");
    }
  }
}
```

Both p1 and p2 refer to **Person** objects that have an idNumber variable instantiated to **318**, a **name** variable instantiated to "**Kostas**" and an **age** variable instantiated to **18**. Although these objects are similar, and from the idNumber one could assume that they are the same person, they are **not** the same. They are two different **Person** objects. Comparing these two objects using the equality operator (==) returns **false**. If the **ObjectEqualityExample** program above is run the output is:

```
p1 is not the same as p2
```

A simple way to compare two objects is to compare all their instance variables or some specific one that identifies an object beyond any reasonable doubt. For example, the **idNumber** of a **Person** object can be assumed to be unique, and so if two **Person** objects have the same **idNumber** they would be equal. As such, instead of comparing **p1** to **p2** by using the equality operator, one could test the equality of the **idNumber** between the two objects as follows:

```
if(p1.idNumber == p2.idNumber) { ... }
```

**Useful Information**: Since one cannot test for equality between two objects by simply using the equality operator (==), but **`Strings`** in Java are objects, it is interesting to look at how one can go about comparing **`Strings`**.

The **`String`** Java class includes a number of methods that help a developer compare the text of the **`String`** object to the text of another **`String`** object. The most helpful method, in our case, for the comparison of two **`String`** objects is the **`.equals()`** method.

For example:

```
…
String s1 = "My name is Markos";
String s2 = "My name is Markos";
System.out.println("s1 is equal to s2:" + s1.equals(s2));
…
```

The output of this program snippet would be:

```
s1 is equal to s2: true
```

Of course, since **`String`** objects are used widely and come bundled with Java, they are sometimes treated in a special manner, which is beyond the scope of this book. Due to that special treatment one may find that under some circumstances the equality operator (==) may be used to effectively compare two **`String`** objects. However, since this is not always the case, it is recommended that the **`.equals()`** method is always used to compare **`Strings`**, in order to avoid any unwanted results.

Abstraction, which was also presented in Topic 4 – Computational Thinking in the Core Computer Science for the IB Diploma Program book[4], relates to the essence of something, without considering specific or unwanted implementation details. For example, the act of double-clicking on an application icon and requesting its launch, is an abstraction. The same action is taken for any application that a user may want to run, but the actual processing that happens behind the scenes (from the input/output devices to memory allocation and processing) is abstracted from the user. In a similar manner, in object-oriented programming, an Abstract Data Type (ADT) is a class that may be used without knowing its implementation details. The ADT describes the data (fields/properties) and operations that can be performed on that data (methods/functions), but hides any details as to how the data is stored and the operations performed. As such, any users that may use the ADT do not know how exactly the ADT works. They are only aware of the data and operations, as well as the expected results of the operations on data.

An important ADT that is extensively used in computer science is the ADT list. The ADT list defines the interface, which means that it defines the methods that a list should encompass to be called a list. A (linear) list is a group of elements arranged in a linear fashion, one after the other. Fundamental operations on the elements of the list include the insertion, deletion and observation of an element, as well as the size of the list. An ADT list may be

implemented in one of two ways; either statically or dynamically. For the static implementation the one-dimensional array may be used, while for the dynamic implementation the single linked list may be used. One-dimensional arrays are linear and allow for the insertion, deletion and observation of their elements, as well as the array size. As such, one-dimensional arrays are a specific implementation of the ADT list. Theoretically an ADT list cannot be filled up. However, arrays, which have to be initialized using a specific finite, number of elements, can and do indeed become full. There are a number of ways to work around this issue. On the other hand, single-linked lists allow for the insertion, deletion and observation of their elements, as well as the list size, and do not fill up. Sections D.4.9 and D.4.10 go into the specifics of static and dynamic ADT list implementations, respectively.

### D.4.8 Describe the applications of lists

| Exit skills. Students should be able to[1]: |
| --- |
| Understand that lists can be used to represent stacks and queues. |

In a list (or a one-dimensional array) any element may be accessed if its index position is known. However, in both stacks and queues, only one specific element may be read, inserted or removed at any given time. Stacks and queues are ADTs that are designed specifically in order to follow these restrictions.



Image D.2: A stack of magazines

**Stacks**

Stacks are Last-In-First-Out (LIFO) structures, meaning that the last element that goes into a stack is the first element to come out. For example, in a stack of plates one may add another plate on top of the stack, but he/she may remove only the topmost plate, which also happens to be the last plate added to the stack. Another example might be a stack of magazines in a waiting room. The people waiting may only remove the magazine at the top of the pile and read it. Anyone finishing reading their magazine will return the magazine to the stack and add it to the top of the pile. As long as no one adds another magazine to the stack, the returned magazine will be the one at the top of the pile.

Figure D.3 visually presents how an empty stack may fill with elements. In this specific example, a magazine is placed on top of the stack every time. Adding elements on top of a

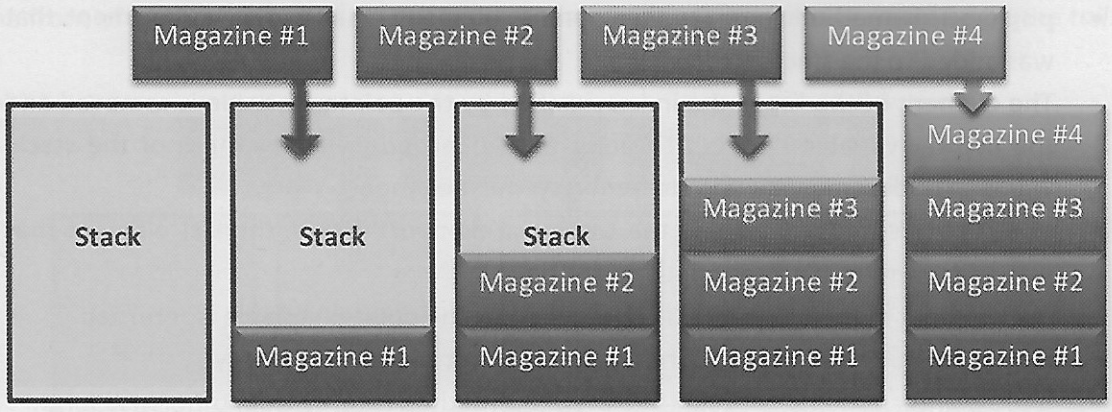



**Figure D.3: Pushing a magazine to the stack**

stack is called *pushing*.

Figure D.4 visually presents how a stack with elements may become empty. In this specific example, a magazine is removed from the top of the stack every time. Removing elements from the top of a stack is called *popping*.

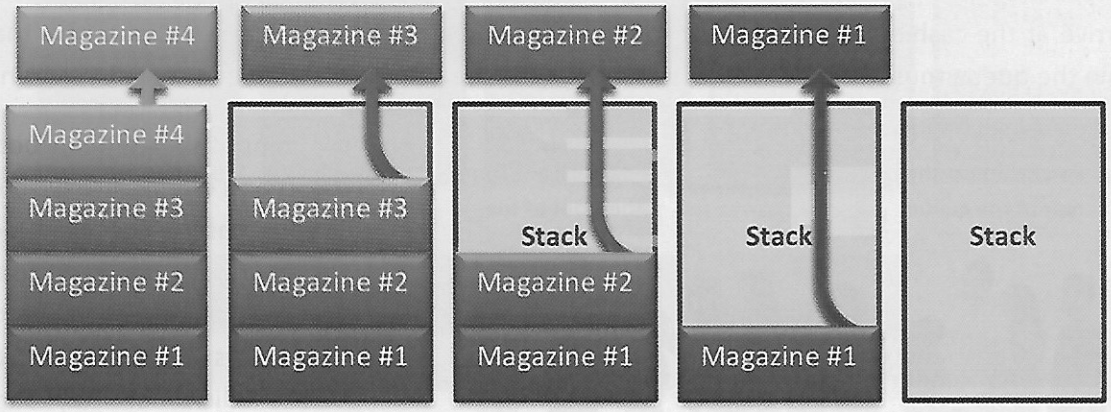



**Figure D.4: Popping a magazine from the stack**

ADT lists described in section D.4.7 can be used to to represent a stack. The fundamental operations of an ADT list upon its elements include the insertion, deletion and observation of an element, as well as the array size. A stack essentially needs the same operations, but with some restrictions so that only the top element may be inserted, deleted or observed. If the stack is empty, an error message should appear if the user tries to delete or observe the top element of the stack. In order to provide this restriction, a pointer variable needs to be in place that will always keep the index of the last element that was added to the stack, and be updated accordingly. This pointer variable will restrict the stack operations to that element.

The main methods of a stack that may be implemented using ADT lists are the following:

- **`push()`**: This method adds an element to the stack as the topmost element. The element is added according to the list index signified by the pointer variable.

- `pop()`: This method removes the topmost element. That is, the last element that was added to the stack.
  The element of the list at the index signified by the pointer variable is removed and the pointer variable updated to point to the second-to-last element of the stack, which then becomes the last element (top-most element).
- `peek()`: This method returns the top-most element, that is, the last element that was added to the stack, but does not remove it.
  The element of the list at the index signified by the pointer variable is returned.
- `isEmpty()`: This method returns whether or not the stack is empty.
  If the pointer variable is equal to -1 then this method returns true, otherwise false.
- `size()`: This method returns the size of the stack. That is, how many elements there are in the stack.
  This method returns the index of the pointer variable plus one.

**Queues**

Queues are lines, like the one that is shown in Image D.3. Queues are First-In-First-Out (FIFO) structures, meaning that the first element that goes into a queue is the first element to come out. For example, when waiting in a queue at the supermarket cashier, the first to arrive at the cashier will be the first to be served and exit the supermarket. Customers that join the queue must join at the end, while the customer that leaves the queue is at the front.

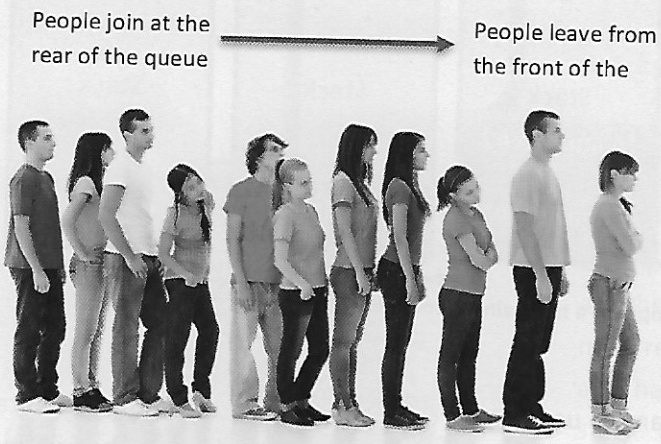



**Image D.3: A queue (line) of people**

In computer science, both stacks and queues are used in various cases. For example, an operating system uses queues to keep track of documents to be printed. Mouse clicks or keyboard strokes are also put in a queue so that the computer system is aware of which clicks or keyboard strokes happened and in what order.

Figure D.5 visually presents how an empty queue may fill with elements. In this specific example, a customer is inserted at the back of the queue every time. Adding elements to the back of the queue may be called *inserting*, *putting*, *adding* or *enqueuing*.

Figure D.6 visually presents how a queue with elements may become empty. In this specific example, a customer is removed from the front of the queue every time. Removing elements from the front of a queue may be called *deleting*, *getting* or *dequeuing*.

The front of a queue may also be the *head* of the queue, while the back may also be the *tail*.

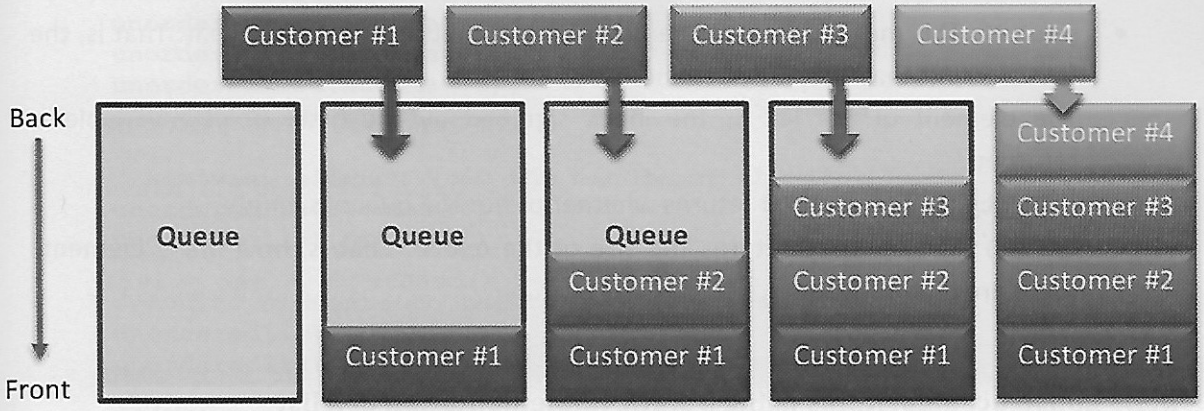



**Figure D.5: Inserting a customer into the queue**

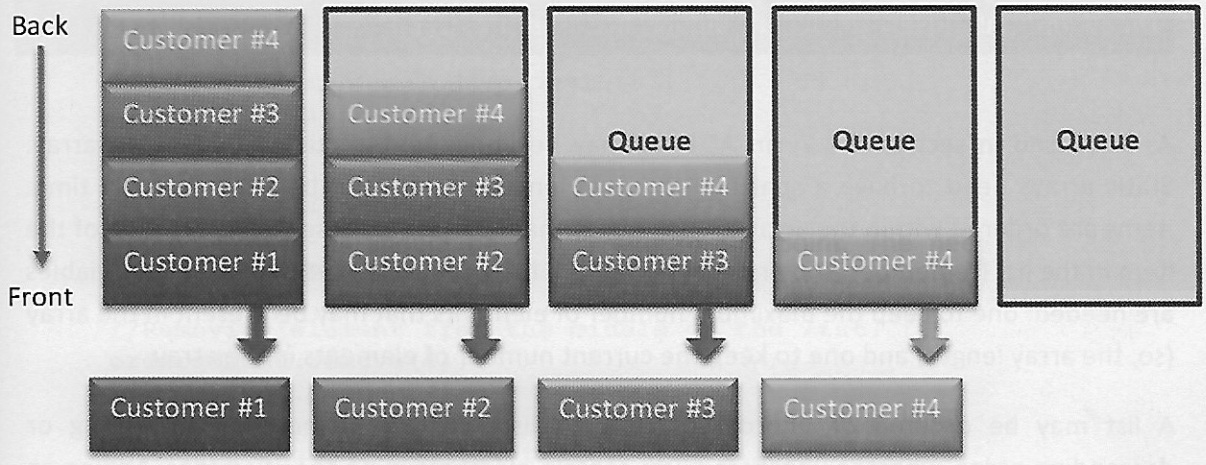



**Figure D.6: Removing a customer from the queue**

ADT lists (described in section D.4.7) can be used to to represent a queue. The fundamental operations of an ADT list on its elements include the insertion, deletion and observation of an element, as well as the array size. A queue essentially needs the same operations, but with some restrictions so that an element may only be inserted at the tail of the queue, while only the element at the head of the queue may be deleted or observed. If the queue is empty, an error message should appear if the user tries to delete or observe the element at the head of the queue. In order to provide this restriction, two pointer variables need to be in place and updated accordingly, that will always keep the index of the head and the tail elements of the queue.

The main methods of a queue that may be implemented using ADT lists are the following:

- **`insert()`**: This method adds an element to the queue as the last element. The element is added according to the list index signified by the tail pointer variable.
- **`remove()`**: This method removes the first element. That is, the first element that was added to the queue.

The element of the list at the index signified by the head pointer variable is removed and the pointer variable updated to point to the second element of the queue, which then becomes the first element.

- **`peek()`**: This method returns the first element, but does not remove it. That is, the first element that was added to the queue.
  The element of the list at the index signified by the head pointer variable is returned.
- **`isEmpty()`**: This method returns whether or not the queue is empty.
- **`size()`**: This method returns the size of the queue. That is, how many elements there are in the stack.

### D.4.9 Construct algorithms using a static implementation of a list

| Exit skills. Students should be able to[1]: |
| --- |
| Implement a list using singly linked types. Methods that should be known are add (head and tail), insert (in order), delete, list, isEmpty, isFull. |

As discussed in section D.4.7, an ADT list may be implemented statically using an array. Static arrays need to have a specific maximum length that cannot be altered at run time. Items are ordered within the array in such a way that its index represents the position of the item in the list (remember, the first element of the list will have an index of 0). Two variables are needed: one to keep the maximum number of elements that may be present in the array (so, the array length) and one to keep the current number of elements in the array.

A list may be ordered or unordered. Ordered lists contain elements in ascending or descending order, whereas unordered lists contain elements in no specific order. Adding an element to an ordered list requires that the element be placed in the correct position so that the order of the list is maintained. On the other hand, any element can be added to an unordered list either from the head or the tail of the list.

The following code represents a static implementation of a list using a static array. The variable **`MAX_LIST`** holds the number of maximum elements that may be present in the array, while the variable **`numItems`** holds the current number of elements in the array. The array in this example is named **`items`** and holds integers (for the sake of simplicity). It could have just as easily held some other primitive type or user-defined object.

```
public class ADTListStatic
{
  private final int MAX_LIST = 50;
  private int numItems;
  private int items[];

  public static void main(String[] args) {
    System.out.println("");
    System.out.println("Unordered list");
    System.out.println("");

    ADTListStatic unorderedlist = new ADTListStatic();
```

```
    //Adding elements to the head of the list.
    unorderedlist.addHead(3);
    unorderedlist.addHead(4);
    unorderedlist.addHead(5);
    unorderedlist.addHead(2);
    unorderedlist.addHead(6);
    unorderedlist.present();

    //Removing element from the middle of the list.
    unorderedlist.remove(4);
    unorderedlist.present();

    //Adding elements to the tail of the list.
    unorderedlist.addTail(1);
    unorderedlist.present();

    System.out.println("");
    System.out.println("Ordered list");
    System.out.println("");

    ADTListStatic orderedlist = new ADTListStatic();

    //Adding elements to the list.
    orderedlist.insert(3);
    orderedlist.insert(4);
    orderedlist.insert(5);
    orderedlist.insert(2);
    orderedlist.insert(6);
    orderedlist.present();

    //Removing element from the middle of the list.
    orderedlist.remove(4);
    orderedlist.present();

    //Adding elements to the tail of the list.
    orderedlist.insert(15);
    orderedlist.present();
  }

  public ADTListStatic() {
    items = new int[MAX_LIST];
    numItems = 0;
  }

  public boolean isEmpty() {
    return numItems == 0;
  }

  public boolean isFull() {
    return numItems == MAX_LIST;
  }

  public int size() {
    return numItems;
  }

  public int peek(int index) {
    if (index >= 0 && index < numItems) {
      return items[index];
    }
```

```
    System.out.println("out of range: 0 - " + (numItems-1));
    return -1;
  }

  /*
   * Method for adding an element to the head of an unordered list.
   * Before adding the element, all the elements to the right
   * of the head are shifted to the right to make way for the
   * new element.
   */
  public void addHead(int item) {
    if (isFull()) {
      System.out.println("List is full.");
    } else {

      int index = 0;
      for (int pos = numItems; pos >= index; pos--) {
        items[pos+1] = items[pos];
      }
      items[index] = item;
      numItems++;
    }
  }

  /*
   * Method for adding an element to the tail of an unordered list.
   * No shifting is necessary in this case.
   */
  public void addTail(int item) {
    if (isFull()) {
      System.out.println("List is full.");
    } else {

      int index = numItems;
      items[index] = item;
      numItems++;
    }
  }

  /*
   * Method for adding an element to a specific index.
   * Before adding the element, all the elements to the right
   * of the specific index are shifted to the right to make
   * way for the new element.
   */
  public void add(int index, int item) {
    if (isFull()) {
      System.out.println("List is full.");
    } else {
      if (index >= 0 && index <= numItems){
        for (int pos = numItems; pos >= index; pos--) {
          items[pos+1] = items[pos];
        }

        items[index] = item;
        numItems++;

      } else {
        System.out.println("out of range: 0 - " + (numItems-1);
      }
```

```
    }
  }

  // Method for adding an element to an ordered list.
  public void insert(int item) {
    if (isFull()) {
      System.out.println("List is full.");
    } else {
      if(isEmpty()) {
        add(0, item);
      } else {
        int index = 0;

        for (int pos = 0; pos < numItems; pos++) {
          index = pos;

          if (item <= items[pos]) //For ascending order.
          if (item <= items[pos]) //For descending order.
          {
            index = pos;
            break;
          } else {
            index = pos+1;
          }
        }

        add(index, item);
      }
    }
  }

  /*
   * Method for removing an element from a specific index.
   * The element is removed by shifting all the elements to
   * the right of the specific index towards the left, to take up
   * the space created by the removed element.
   */
  public void remove(int index) {
    if (index >=0 && index < numItems) {
      for (int pos=index+1; pos<=size(); pos++) {
        items[pos-1] = items[pos];
      }
      numItems--;
    } else {
      System.out.println("out of range: 0 - " + (numItems-1));
    }
  }

  public void present() {

    System.out.println("List elements:");

    for (int pos = 0; pos < numItems; pos++) {
      System.out.print(items[pos] + " ");
    }

    System.out.println();
  }
}
```

The above static implementation of a list using a static array includes the following methods:

- **`ADTListStatic()`**: This method is the constructor. It initializes the array to be used as the list with **`MAX_LIST`** positions. It also initializes the variable **`numItems`**, which counts the number of elements in the list, to zero.
- **`boolean isEmpty()`**: This method checks whether the list is empty by comparing **`numItems`** to zero.
- **`boolean isFull()`**: This method checks whether the list is full by comparing **`numItems`** to **`MAX_LIST`**.
- **`int size()`**: This method just returns **`numItems`**, which holds the number of items in the list.
- **`int peek(int index)`**: This method returns the element of the list at the specified index. If the index provided is outside the bounds of the list then an error message appears.
- **`void addHead(int item)`**: This method first checks whether or not the list is full. Then it shifts all the elements one position to the right and adds the element provided to the beginning of the list.
- **`void addTail(int item)`**: This method first checks whether or not the list is full. It then adds the element provided to the end of the list.
- **`void insert(Node node)`**: This method adds the **`Node`** provided to the correct position in the list in order to maintain list order (ascending or descending).
- **`void add(int index, int item)`**: This method is a **`private`** method that adds an element to a specific index.
- **`void remove(int index)`**: This method shifts all the elements to the right of the specified index one position to the left, so as to take up the space of the element that needs to be removed.
- **`void present()`**: This method just prints out the list, using a **`for`** loop that runs from zero to one less than **`numItems`**.
- **`void main(String[] args)`**: This method is added for testing purposes. It creates a new **`ADTListStatic`** object and adds numbers **`3`**, **`4`**, **`5`**, **`2`**, **`6`** to the head of the list. It then presents the list, which will be the following:

  ```
  6 2 5 4 3
  ```

  It then removes the element at index **`4`** and once again presents the list (remember, the first element of the list has an index of zero):

  ```
  6 2 5 4
  ```

  Finally the method adds number **`1`** to the tail of the list and presents the following output:

  ```
  6 2 5 4 1
  ```

  The main method then creates a new **`ADTListStatic`** object **`(listOrdered)`** and adds numbers **`3`**, **`4`**, **`5`**, **`2`**, **`6`** using the insert method. This way the list remains ordered. It then presents the list, which will be the following:

  ```
  2 3 4 5 6
  ```

It then removes the element at index **4** and once again presents the ordered list (remember, the first element of the list has an index of zero):

```
2 3 4 5
```

Finally, the method adds number **15** to the ordered list and presents the following output:

```
2 3 4 5 15
```

### Stacks using a static implementation of a list

Remember, stacks are Last-In-First-Out (LIFO) structures, meaning that the last element that goes into a stack is the first element to come out. A stack needs the same operations as the ADT list, but with some restrictions so that just the top element may be altered or examined.

For the sake of simplicity, a stack that may include integers will be presented in this section. Of course, the algorithms could easily be altered for the stack to include some other primitive data type or even user-defined objects.

The main methods of a stack that may be implemented using ADT lists are the following: **`void push(int item)`**, **`int pop()`**, **`int peek()`**, **`boolean isEmpty()`** and **`int size()`**.

The implementation of a stack using static arrays is quite simple:

```
public class ADTListStaticStack
{
  private ADTListStatic list;

  public ADTListStaticStack() {
    list = new ADTListStatic();
  }

  public void push(int item) {
    list.addHead(item);
  }

  public int pop() {
    int item = peek();
    list.remove(0);
    return item;
  }

  public int peek() {
    return list.peek(0);
  }

  public boolean isEmpty() {
    return list.isEmpty();
  }

  public int size() {
    return list.size();
  }
}
```

Pushing an element to the stack involves just adding the element to the head of the list. Popping the element from the stack involves peeking at the top element of the list, storing it in a variable, removing the element from the list and returning the value in the variable. So, popping an element is different from removing an element from the stack in that popping also returns the value of the element. Peeking at the element at the head of the stack involves just peeking at the head of the list. Finding the size of the stack and whether or not it's empty involves simply calling the respective list methods and returning the results.

The following example could be used to test out the `ADTListStaticStack`:

```
public static void main(String[] args) {
  ADTListStaticStack stack = new ADTListStaticStack();

  stack.push(5);
  stack.push(4);
  stack.push(3);
  stack.push(2);
  stack.push(1);

  System.out.println("Top element is: " + stack.peek());
  System.out.println("Popping top element: " + stack.pop());
  System.out.println("Top element is now: " + stack.peek());
}
```

A new stack is created and the numbers 5, 4, 3, 2, 1 are pushed giving the following stack:

```
Head 1 2 3 4 5 Tail
```

The top element is printed using the `peek()` method, then popped, using the `pop()` method, and then the new top element is printed again, giving the following output:

```
Top element is: 1
Popping top element: 1
Top element is now: 2
```

### Queues using a static implementation of a list

Remember, queues are First-In-First-Out (FIFO) structures, meaning that the first element that goes into a queue is the first element to come out. A queue needs the same operations as the ADT list, but with some restrictions so that an element may only be inserted at the tail of the queue, while only the element at the head of the queue may be deleted or observed.

For the sake of simplicity, a queue that may include integers will be presented in this section. Of course, the algorithms could easily be altered for the queue to include some other primitive data type, or even user-defined objects.

The main methods of a queue that may be implemented using ADT lists are the following: `void insert(int item)`, `int remove()`, `int peek()`, `boolean isEmpty()` and `int size()`.

The implementation of a queue using static arrays is quite simple:

```
public class ADTListStaticQueue
{
  private ADTListStatic list;

  public ADTListStaticQueue() {
    list = new ADTListStatic();
  }

  public void insert(int item) {
    list.addTail(item);
  }

  public int remove() {
    int item = peek();
    list.remove(0);
    return item;
  }

  public int peek() {
    return list.peek(0);
  }

  public boolean isEmpty() {
    return list.isEmpty();
  }

  public int size() {
    return list.size();
  }
}
```

Inserting an element into the queue involves adding the element to the tail of the list. Removing the first element of the queue involves peeking at the first element of the list, storing it in a variable, removing the element from the list and returning the value in the variable. Peeking at the first element of the queue involves just peeking at the first element of the list. Finding the size of the queue and whether or not it's empty involves simply calling the respective list methods and returning the results.

The following example could be used to test out the **ADTListStaticQueue**:

```
public static void main(String[] args) {
  ADTListStaticQueue queue = new ADTListStaticQueue();

  queue.insert(5);
  queue.insert(4);
  queue.insert(3);
  queue.insert(2);
  queue.insert(1);

  System.out.println("First element is: " + queue.peek());
  System.out.println("Removing first element: " + queue.remove());
  System.out.println("First element is now: " + queue.peek());
}
```

A new queue is created and the numbers **5, 4, 3, 2, 1** are inserted giving the following queue:

```
Front 5 4 3 2 1 Back
```

The front element is printed using the **peek()** method, then removed, using the **remove()** method, and then the new front element is printed again, giving the following output:

```
First element is: 5
Removing first element: 5
First element is now: 4
```

### D.4.10 Construct list algorithms using object references

| Exit skills. Students should be able to[1]: |
| --- |
| Implement a list using singly-linked types. Methods that should be known are add (head and tail), insert (in order), delete, list, isEmpty, isFull. |

As discussed in section D.4.7, an ADT list may be implemented dynamically using a linked list. In section D.4.9 an ADT list was implemented using static arrays. Static arrays need to have a specific maximum length that may not be altered at run time. That means that although, in theory, due to array implementation restrictions, ADT lists do not have a maximum number of elements that they can hold, they do have a maximum number of elements that they can hold when implemented in a static manner, using static arrays. Dynamic implementation of ADT lists overcomes these limitations.

A linked list is made up of two parts: the head (a.k.a. reference variable) and the nodes. The head includes a variable that points to the first item of the list, as well as, sometimes, another variable that holds the number of items currently in the list. The nodes are the elements of the list. They include a data variable (that holds some kind of data, from primitive types to user defined objects) and another variable that points and links to the next node in the list.

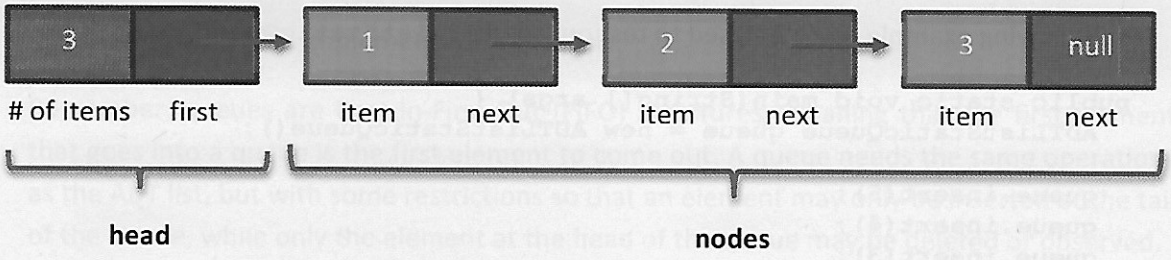



**Figure D.7: An example of a linked list with three nodes**

Figure D.7 displays an example of a linked list with three nodes that hold integers. The head of the list holds the number of elements in the list (i.e. three), as well as a link to the first node of the list. The first node of the list includes some data (i.e. 1), as well as a link to the next (second) node of the list. The second node of the list includes some data (i.e. 2), as well as a link to the next (third) node of the list. The third, and last, node of the list includes some

data (i.e. 3), but does not link to another node, since there is no other (fourth) node. As such, the value of the link is null.

Figure D.8 displays an example of a linked list where the pointer of the head's first variable is null. In this case, the head does not point to t of the list, as there is no first

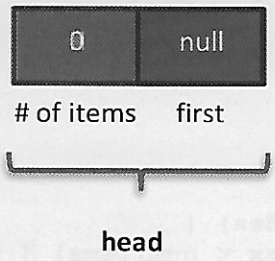



**Figure D.8: An example of an empty linked list**

A linked list may be ordered or unordered. Ordered lists contain elements in ascending or descending order, whereas unordered lists contain elements in no specific order. Adding an element to an ordered list requires that the element be placed in the correct position so that the order of the list is maintained. On the other hand, any element can be added to an unordered list either from the head or the tail of the list.

The code below represents a dynamic implementation of a list using a linked list. Two classes are needed:

- **Node** class: Represents a node of the linked list. It includes a data variable named **item** (that holds an integer for the sake of simplicity) and another variable, **next**, that points and links to the next node in the list.
- **ADTListHead** class: Represents the head of the linked list. It includes a variable, **numItems**, that holds the number of elements in the list, as well as a variable, **first**, that links to the first **Node** object of the list. If there is no first node, the list is empty and the **first** variable is **null**.

```
public class Node {
  private int item;
  private Node next;

  public Node(int item) {
    this.item = item;
  }

  public void setItem(int item) {
    this.item = item;
  }

  public int getItem() {
    return item;
  }

  public void setNext(Node next) {
    this.next = next;
```

```
  }

  public Node getNext() {
    return next;
  }
}

public class ADTListHead {
  private Node first;
  private int numItems;

  public ADTListHead() {
    first = null;
    numItems = 0;
  }

  public Node find(int index) {
    if (index >= 0 && index < numItems) {
      Node current = first;
      for(int i=0; i<index; i++) {
        current = current.getNext();
      }
      return current;
    }

    System.out.println("out of range: 0 - " + (numItems-1));
    return null;
  }

  //Methods for adding elements to an unordered list.
  public void addHead(Node node) {
    node.setNext(first);
    first = node;
    numItems++;
  }

  public void addTail(Node node) {
    if (size() == 0) {
      node.setNext(first);
      first = node;
    } else {
      Node last = find(size()-1);
      last.setNext(node);
    }
    numItems++;
  }

  // Methods for adding elements to an ordered list.
  public void insert(Node node) {
    Node current = first;

    //For ascending order.
    if (size() == 0 || current.getItem() >= node.getItem())
    //For descending order.
    if (size() == 0 || current.getItem() <= node.getItem())
    {
      node.setNext(first);
      first = node;

    } else {
```

```
    //For ascending order.
    while (current.getNext() != null &&
    current.getNext().getItem() < node.getItem())
    //For descending order.
    while (current.getNext() != null &&
    current.getNext().getItem() > node.getItem())
    {
      current = current.getNext();
    }

    node.setNext(current.getNext());
    current.setNext(node);
  }

  numItems++;
}

public int peek(int index) {
  Node node = find(index);
  return node.getItem();
}

public void remove(int index) {
  Node nodeToRemove = find(index);

  if (index == 0) {
    first = nodeToRemove.getNext();

  } else {
    Node previousNode = find(index-1);

    if (index == size()-1) {
      previousNode.setNext(null);
    } else {
      previousNode.setNext(nodeToRemove.getNext());
    }
  }
    numItems--;
}

public boolean isEmpty() {
  return first == null;
}

public int size() {
  return numItems;
}

public void present() {
  System.out.println("List elements:");

  Node current = first;

  while (current != null) {
    System.out.print(current.getItem() + " ");
    current = current.getNext();
  }

  System.out.println();
}
```

```
  public static void main(String[] args) {
    System.out.println("");
    System.out.println("Unordered list");
    System.out.println("");

    ADTListHead listUnordered = new ADTListHead();

    //Adding elements to the head of the list.
    listUnordered.addHead(new Node(3));
    listUnordered.addHead(new Node(4));
    listUnordered.addHead(new Node(5));
    listUnordered.addHead(new Node(2));
    listUnordered.addHead(new Node(6));
    listUnordered.present();

    //Removing element from the middle of the list.
    System.out.println("Remove element at index 4.");
    listUnordered.remove(4);
    listUnordered.present();

    //Adding elements to the tail of the list.
    System.out.println("At element 1 at the tail.");
    listUnordered.addTail(new Node(1));
    listUnordered.present();

    //Peek at the 2nd node's data.
    int secondNodeData = listUnordered.peek(1);
    System.out.println("Second node's data: " + secondNodeData);

    System.out.println("");
    System.out.println("Ordered list");
    System.out.println("");

    ADTListHead listOrdered = new ADTListHead();

    //Adding elements to the list.
    listOrdered.insert(new Node(3));
    listOrdered.insert(new Node(4));
    listOrdered.insert(new Node(5));
    listOrdered.insert(new Node(2));
    listOrdered.insert(new Node(6));
    listOrdered.present();

    //Removing element from the middle of the list.
    System.out.println("Remove element at index 4.");
    listOrdered.remove(4);
    listOrdered.present();

    //Adding elements to the tail of the list.
    System.out.println("Add element 15.");
    listOrdered.insert(new Node(15));
    listOrdered.present();
  }
}
```

The above dynamic implementation of a list using a linked list includes the following methods:

- **`ADTListHead()`**: This method is the *constructor*. It initializes the variable **`first`** to **`null`** and also initializes the variable **`numItems`**, which counts the number of elements in the list, to zero.
- **`boolean isEmpty()`**: This method checks whether the list is empty by checking whether **`first`** is **`null`**.
- **`int size()`**: This method just returns **`numItems`**, which holds the number of items in the list.
- **`int peek(int index)`**: This method returns the data of the node at the specified index. If the index provided is outside the bounds of the list, then an error message appears.
- **`void addHead(Node node)`**: This method adds the **`Node`** provided to the beginning of the list.
- **`void addTail(Node node)`**: This method adds the **`Node`** provided to the end of the list.
- **`void insert(Node node)`**: This method adds the **`Node`** provided to the correct position in the list in order to maintain list order (ascending or descending).
- **`void remove(int index)`**: This method removes the **`Node`** at the specified index. For that to happen, the previous node in the list is found and its **`Next`** variable is set to point to the **`Node`** that comes after the one to be removed, as shown in Figure D.9. Two special cases exist:

  1. If the node to be removed is the first node in the list, then the **`first`** variable of the **`ADTListHead`** object is changed so that it points to the second node in the list.

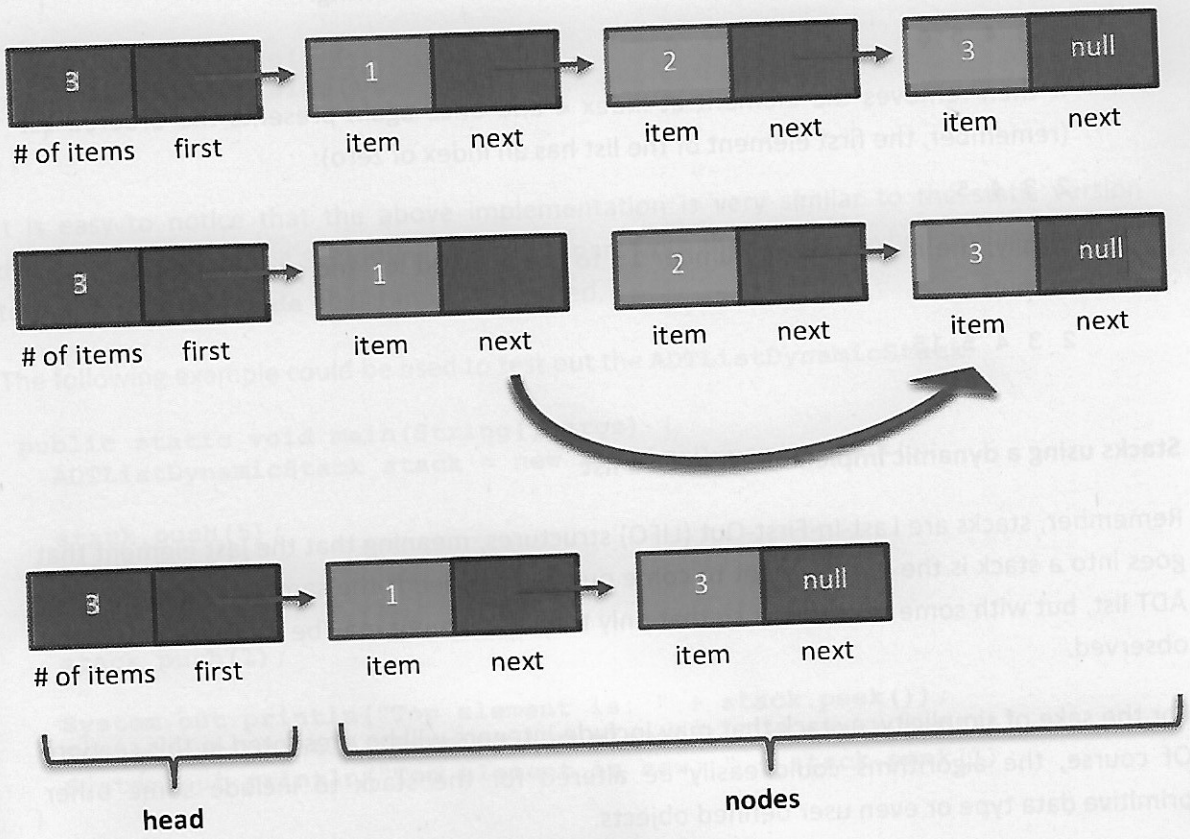



**Figure D.9: Deleting a linked list node**

2. If the node to be removed is the last node in the list, then the **next** variable of the second to last node is set to **null**.

- **void present()**: This method just prints out the list. It uses a **while** loop until the **null** item is found as the **next** node, meaning that there is no next node and as such the end of the list has been reached.
- **void main(String[] args)**: This method is added for testing purposes. It creates a new **ADTListHead** object (**listUnordered**) and adds numbers **3**, **4**, **5**, **2**, **6** to the head of the unordered list. It then presents the unordered list, which will be the following:

  ```
  6 2 5 4 3
  ```

  It then removes the element at index **4** and once again presents the unordered list (remember, the first element of the list has an index of zero):

  ```
  6 2 5 4
  ```

  The method adds number **1** to the tail of the unordered list and presents the following output:

  ```
  6 2 5 4 1
  ```

  Finally, the method peeks at the second node's data and prints it out, as follows:

  ```
  Second node's data: 2
  ```

  The main method then creates a new **ADTListHead** object **(listOrdered)** and adds numbers **3**, **4**, **5**, **2**, **6** using the insert method. This way the list remains ordered. It then presents the list, which will be the following:

  ```
  2 3 4 5 6
  ```

  It then removes the element at index **4** and once again presents the ordered list (remember, the first element of the list has an index of zero):

  ```
  2 3 4 5
  ```

  Finally, the method adds number **15** to the ordered list and presents the following output:

  ```
  2 3 4 5 15
  ```

### Stacks using a dynamic implementation of a list

Remember, stacks are Last-In-First-Out (LIFO) structures, meaning that the last element that goes into a stack is the first element to come out. A stack needs the same operations as the ADT list, but with some restrictions so that only the top element may be inserted, deleted or observed.

For the sake of simplicity, a stack that may include integers will be presented in this section. Of course, the algorithms could easily be altered for the stack to include some other primitive data type or even user defined objects.

The main methods of a stack that may be implemented using ADT lists are the following: `void push(int item)`, `int pop()`, `int peek()`, `boolean isEmpty()` and `int size()`.

The implementation of a stack using linked lists is quite simple:

```
public class ADTListDynamicStack
{
  private ADTListHead list;

  public ADTListDynamicStack() {
    list = new ADTListHead();
  }

  public void push(int item) {
    list.addHead(new Node(item));
  }

  public int pop() {
    int item = peek();
    list.remove(0);
    return item;
  }

  public int peek() {
    return list.peek(0);
  }

  public boolean isEmpty() {
    return list.isEmpty();
  }

  public int size() {
    return list.size();
  }
}
```

It is easy to notice that the above implementation is very similar to the static version discussed in section D.4.9. Nothing changes, apart from the fact that when pushing an item to the stack, a new `Node` object must be created.

The following example could be used to test out the `ADTListDynamicStack`:

```
public static void main(String[] args) {
  ADTListDynamicStack stack = new ADTListDynamicStack();

  stack.push(5);
  stack.push(4);
  stack.push(3);
  stack.push(2);
  stack.push(1);

  System.out.println("Top element is: " + stack.peek());
  System.out.println("Popping top element: " + stack.pop());
  System.out.println("Top element is now: " + stack.peek());
}
```

A new stack is created and the numbers 5, 4, 3, 2, 1 are pushed, giving the following stack:

```
Head 1 2 3 4 5 Tail
```

The top element is printed using the `peek()` method, then popped, using the `pop()` method. Then the new top element is printed again, giving the following output:

```
Top element is: 1
Popping top element: 1
Top element is now: 2
```

### Queues using a dynamic implementation of a list

Remember, queues are First-In-First-Out (FIFO) structures, meaning that the first element that goes into a queue is the first element to come out. A queue needs the same operations as the ADT list, but with some restrictions so that an element may only be inserted at the tail of the queue, while only the element at the head of the queue may be deleted or observed.

For the sake of simplicity, a queue that may include integers will be presented in this section. Of course, the algorithms could easily be altered for the queue to include some other primitive data type or even user-defined objects.

The main methods of a queue that may be implemented using ADT lists are the following: `void insert(int item)`, `int remove()`, `int peek()`, `boolean isEmpty()` and `int size()`.

The implementation of a queue using linked lists is quite simple:

```
public class ADTListDynamicQueue
{
  private ADTListHead list;

  public ADTListDynamicQueue() {
    list = new ADTListHead();
  }

  public void insert(int item) {
    list.addTail(new Node(item));
  }

  public int remove() {
    int item = peek();
    list.remove(0);
    return item;
  }

  public int peek() {
    return list.peek(0);
  }

  public boolean isEmpty() {
    return list.isEmpty();
  }

  public int size() {
```

```
        return list.size();
    }
}
```

It is easy to notice that the above implementation is very similar to the static version discussed in section D.4.9. Nothing changes apart from the fact that when inserting an item to the queue a new **Node** object must be created.

The following example could be used to test out the **ADTListStaticQueue**:

```
public static void main(String[] args) {
    ADTListDynamicQueue queue = new ADTListDynamicQueue();

    queue.insert(5);
    queue.insert(4);
    queue.insert(3);
    queue.insert(2);
    queue.insert(1);

    System.out.println("First element is: " + queue.peek());
    System.out.println("Removing first element: " + queue.remove());
    System.out.println("First element is now: " + queue.peek());
}
```

A new queue is created and the numbers **5**, **4**, **3**, **2**, **1** are inserted giving the following queue:

```
Front 5 4 3 2 1 Back
```

The front element is printed using the **peek()** method, then removed, using the **remove()** method. Then the new front element is printed again, giving the following output:

```
First element is: 5
Removing first element: 5
First element is now: 4
```

### D.4.11 Construct algorithms using the standard library collections included in JETS

| Exit skills. Students should be able to[1]: |
| --- |
| Implement algorithms using the classes ArrayList and LinkedList. Students should have a broad understanding of the operation of these lists and their interface (methods) but not of the internal structure. |

Common programming tasks that will most likely be needed by any kind of project are usually packed in libraries. These libraries can be loaded by a programmer so that he/she may take advantage of any programming tasks they include and avoid rewriting them for his/her project. Libraries allow for various programming tasks to be written once and then easily reused whenever necessary. Furthermore, bugs are avoided as known working libraries that have been carefully tested may be used as "black boxes". Most programming

languages include a standard library, which includes the programming tasks most widely used. Java's standard library collection included in JETS encompasses, amongst others, **ArrayList** and **LinkedList** classes.

Classes (a.k.a. package members) from standard library collections may be imported using the **import** statement at the top of the file in which the classes are to be used, as such:

```
import java.util.ArrayList;
```

After that, one can create **ArrayList** objects. The following example creates two **ArrayList** objects, **list1** and **list2** respectively, that will hold **Integer** objects.

```
//Creates an ArrayList object.
ArrayList<Integer> list1 = new ArrayList<Integer>();

//Creates an ArrayList object with an initial capacity
//of 10 elements.
ArrayList<Integer> list2 = new ArrayList<Integer>(10);
```

ArrayLists may include only objects as elements and so do not support primitive types. They can, however, include any number of elements. Even if an initial capacity has been determined, it can change during run time without the programmer noticing. So, compared to static arrays, ArrayLists may include any number of elements.

A full list of all the methods included in the **ArrayList** class can be found in the official Java documentation[5]. However, the most common methods are described below:

- Adding elements to an **ArrayList**:
  - **boolean add(Element e)**: Adds element **e** to the end of the list. Element **e** may be any kind of object, even user defined objects. It returns true if the element is added successfully.
  - **void add(int index, Element e)**: Adds element **e** to the list at the position specified by **index**. Shifts the element at the specified **index** and all subsequent elements to the right.
- Removing elements from an **ArrayList**:
  - **void clear()**: Removes all the elements from the list.
  - **E remove(int index)**: Removes the element at the specified **index**. All elements to the right of the **index** are shifted to the left, to take up the space of the removed element. The element removed is returned from this method. The return type, **E**, may be any kind of object, even user defined objects.
  - **protected void removeRange(int start, int end)**: Removes the elements between the start (inclusive) and end (exclusive) indexes. Shifts to the left any elements to the right of, and including, the end index.
- Retrieving elements from an **ArrayList**:

[5] Class ArrayList. In *Java Documentation*. Retrieved 19:00, July 8, 2016, https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

- `E get(int index)`: Returns the element at the specified `index`. The return type, `E`, may be any kind of object, even user-defined objects.
- `<T>T[] toArray(T[] a)`: Populates a static array `a` with all the elements in the list in proper sequence.

- Setting an element in an `ArrayList`:
  - `E set(int index, E element)`: Replaces the element at the specified `index` with the specified element `E`. The element previously occupying the specific `index` is returned.
- Searching for an element in an `ArrayList`:
  - `boolean contains(Object o)`: Returns true if the specified object `o` is found in the list.
  - `int indexOf(Object o)`: Returns the index of the first occurrence of the specified object `o` in the list. If `o` is not in the list, then `-1` is returned.
  - `int lastIndexOf(Object o)`: Returns the index of the last occurrence of the specified object `o` in the list. If `o` is not in the list, then `-1` is returned.
- Iterating an `ArrayList`:
  - `Iterator iterator()`: Returns an iterator over the elements in the list.
  - `ListIterator listIterator()`: Returns a list iterator over the elements in the list.
- Checking whether an `ArrayList` is empty:
  - `boolean isEmpty()`: Returns whether the list is empty or not.
- Getting the size of an `ArrayList`:
  - `int size()`: Returns the number of elements in the list.

**Useful Information**: Iterators are the easiest way one can cycle through the elements of a collection, including `ArrayList` and `LinkedList`. There exist two types of iterators:

- **Iterator**: allows programmers to cycle through a collection, as well as remove any elements they may want.
  - An iterator, initialized to the beginning of the collection, is obtained by calling the collection's `iterator()` method.
  - Iterator methods:
    - `boolean hasNext()`: Returns true if there are more elements in the collection.
    - `Object next()`: Returns the next element.
    - `void remove()`: Removes the current element.
- **ListIterator**: extends the Iterator to allow bidirectional traversal of a collection, as well as additional modification to its structure and elements, including addition of objects to the collection.
  - A list iterator, initialized to the beginning of the collection, is obtained by calling the collection's `listIterator()` or `listIterator(int index)` methods.
  - List iterator methods:
    - `boolean hasNext()`: Returns true if there are more elements in the collection.
    - `boolean hasPrevious`: Returns true if there is a previous element.
    - `Object next()`: Returns the next element.
    - `Object previous()`: Returns the previous element.
    - `int nextIndex()`: Returns the index of the next element. If there is not a next element, it returns the size of the list.
    - `int previousIndex()`: Returns the index of the previous element. If there is not a previous element, it returns `-1`.
    - `void remove()`: Removes the current element.
    - `void set(Object o)`: Assigns `o` to the current element, which is the element returned by a call to either `next()` or `previous()`.

Both `Iterator` and `ListIterator` classes are part of the `java.util` standard library collection. As such, in order to use `ArrayList`, `Iterator` and `ListIterator` for our example, the following import statements need to be present at the top of the file:

```
import java.util.Iterator;
import java.util.ListIterator;
import java.util.ArrayList;
```

**Useful Information cont.**: In order to avoid writing many import statements at the top of each file, one may use the asterisk (*) symbol to import all package members as follows:

```
import java.util.*;
```

All package members under the **java.util**, including **Iterator**, **ListIterator** and **ArrayList** are imported using a single statement.

The following example illustrates how an **Iterator** may be used to cycle through all the elements of a collection. Furthermore, it depicts how a **ListIterator** may be used to modify an object of a collection, as well as how it can be used to cycle through all the elements backwards.

```
import java.util.*;

public class IteratorExample {
  public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    Integer one = new Integer(1);
    Integer two = new Integer(2);
    Integer three = new Integer(3);
    list.add(one);
    list.add(two);
    list.add(three);

    //Iterator example.
    Iterator<Integer> iterator = list.iterator();
    while(iterator.hasNext()) {
      System.out.print(iterator.next() + " ");
    }
    System.out.println();

    //List iterator example.
    ListIterator<Integer> literator = list.listIterator();
    while(literator.hasNext()) {
      Integer e = literator.next();
      literator.set(e.intValue()+10);
    }

    while(literator.hasPrevious()) {
      System.out.print(literator.previous() + " ");
    }
    System.out.println();
  }
}
```

The output of the above program is the following:

```
1 2 3
13 12 11
```

The LinkedList class from standard library collections may be imported using the **import** statement at the top of the file in which the class is to be used, as such:

```
import java.util.LinkedList;
```

After that, one can create **LinkedList** objects. The following example creates a **LinkedList** object, **list**, that will hold **Integer** objects.

```
//Creates a LinkedList object.
LinkedList<Integer> list = new LinkedList<Integer>();
```

LinkedLists may include only objects as elements and do not support primitive types. They can, however, include any number of elements. Compared to static arrays, LinkedLists may include any number of elements, just like ArrayLists.

A full list of all the methods included in the **LinkedList** class can be found in the official Java documentation[6]. However, the most common methods are described below:

- Adding elements to a **LinkedList**:
  - **boolean add(Element e)**: Adds element **e** to the end of the list. Element **e** may be any kind of object, even user defined objects.
  - **void add(int index, Element e)**: Adds element **e** to the list at the position specified by **index**. Shifts the element at the specified **index** and all subsequent elements to the right.
  - **void push(Element e)**: Pushes (adds) element **e** to the beginning (head) of the list.
- Removing elements from a **LinkedList**:
  - **void clear()**: Removes all the elements from the list.
  - **E remove(int index)**: Removes the element at the specified **index**. All elements to the right of the **index** are shifted to the left, to take up the space of the removed element. The element removed is returned from this method. The return type, **E**, may be any kind of object, even user defined objects.
  - **E pop()**: Pops (removes) the first (head) element from the list and returns it.
- Retrieving elements from a **LinkedList**:
  - **E peek()**: Returns the first (head) element of the list.
  - **E get(int index)**: Returns the element at the specified **index**. The return type, **E**, may be any kind of object, even user defined objects.
  - **<T>T[] toArray(T[] a)**: Populates a static array **a** with all the elements in the list in proper sequence.
- Setting an element in a **LinkedList**:
  - **E set(int index, E element)**: Replaces the element at the specified **index** with the specified element **E**. The element previously occupying the specific **index** is returned.

---

[6] Class LinkedList. In *Java Documentation*. Retrieved 19:00, July 8, 2016, https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

- Searching for an element in a **LinkedList**:
  - **boolean contains(Object o)**: Returns true if the specified object **o** is found in the list.
  - **int indexOf(Object o)**: Returns the index of the first occurrence of the specified object **o** in the list. If **o** is not in the list, then **-1** is returned.
  - **int lastIndexOf(Object o)**: Returns the index of the last occurrence of the specified object **o** in the list. If **o** is not in the list, then **-1** is returned.
- Iterating the **LinkedList**:
  - **Iterator descendingIterator()**: Returns an iterator over the elements in the list from last (tail) to first (head).
  - **ListIterator listIterator()**: Returns a list iterator over the elements in the list.
- Checking whether a **LinkedList** is empty:
  - **boolean isEmpty()**: Returns whether the list is empty or not.
- Getting the size of a **LinkedList**:
  - **int size()**: Returns the number of elements in the list.

As it is easily apparent, both **ArrayList** and **LinkedList** classes have great similarities. Both are used to implement list structures, and both may be used in real world applications without the programmer having to think about what is happening in the internal structure of the classes ("under-the-hood"). Furthermore, both have more or less the same methods. Small differences exist, such as the fact that the **E get(int index)** method is faster in the **ArrayList** class, whereas the **void add(int index, Element e)** is faster in the **LinkedList** class. These small differences are negligible and beyond the scope of this book.

**Useful Information**: Two small programs are included in Appendix A, that present how stacks and queues may be implemented using the **ArrayList** class. These programs also include a text user interface.

### D.4.12 Trace algorithms using the implementations described in assessment statements D.4.9-D.4.11

| Exit skills. Students should be able to[1]: |
| --- |
| Trace algorithms. In examination questions, definitions of ArrayList and LinkedList methods will be given when necessary. |

The algorithms described and implemented in section D.4.9 used a static implementation of a list. The following two **main** methods are traced, which are similar to those of section D.4.9, and make use of **ADTListStaticStack** and **ADTListStaticQueue** objects respectively.

```
public static void main(String[] args) {
  ADTListStaticStack stack = new ADTListStaticStack();

  for(int i=5; i>0; i--) {
    stack.push(i);
  }

  System.out.println("Top element is: " + stack.peek());
  System.out.println("Popping top element: " + stack.pop());
  System.out.println("Top element is now: " + stack.peek());
}
```

| i | i>0 | stack | output |
|---|---|---|---|
| 5 | true | [5] | |
| 4 | true | [4,5] | |
| 3 | true | [3,4,5] | |
| 2 | true | [2,3,4,5] | |
| 1 | true | [1,2,3,4,5] | |
| 0 | false | | |
| | | | Top element is: 1 |
| | | | Popping top element: 1 |
| | | | Top element is now: 2 |

Table D.2: Trace table for static stack example

```
public static void main(String[] args) {
  ADTListStaticQueue queue = new ADTListStaticQueue();

  for(int i=5; i>0; i--) {
    queue.insert(i);
  }

  System.out.println("First element is: " + queue.peek());
  System.out.println("Removing first element: " + queue.remove());
  System.out.println("First element is now: " + queue.peek());
}
```

| i | i>0 | queue | output |
|---|---|---|---|
| 5 | true | [5] | |
| 4 | true | [5,4] | |
| 3 | true | [5,4,3] | |
| 2 | true | [5,4,3,2] | |
| 1 | true | [5,4,3,2,1] | |
| 0 | false | | |
| | | | First element is: 5 |
| | | | Removing first element: 5 |
| | | | First element is now: 4 |

Table D.3: Trace table for static queue example

The algorithms described and implemented in section D.4.10 used object references for the implementation of a list. The following two **main** methods are traced, which are similar to those of section D.4.10, and make use of **ADTListDynamicStack** and **ADTListDynamicQueue** objects respectively.

```
public static void main(String[] args) {
  ADTListDynamicStack stack = new ADTListDynamicStack();

  for(int i=5; i>0; i--) {
```

```
    stack.push(i);
  }

  System.out.println("Top element is: " + stack.peek());
  System.out.println("Popping top element: " + stack.pop());
  System.out.println("Top element is now: " + stack.peek());
}
```

| i | i>0 | stack | output |
|---|---|---|---|
| 5 | true | 5 | |
| 4 | true | 4,5 | |
| 3 | true | 3,4,5 | |
| 2 | true | 2,3,4,5 | |
| 1 | true | 1,2,3,4,5 | |
| 0 | false | | |
| | | | Top element is: 1 |
| | | | Popping top element: 1 |
| | | | Top element is now: 2 |

Table D.4: Trace table for dynamic stack example

```
public static void main(String[] args) {
  ADTListDynamicQueue queue = new ADTListDynamicQueue();

  for(int i=5; i>0; i--) {
    queue.insert(i);
  }

  System.out.println("First element is: " + queue.peek());
  System.out.println("Removing first element: " + queue.remove());
  System.out.println("First element is now: " + queue.peek());
}
```

| i | i>0 | queue | output |
|---|---|---|---|
| 5 | true | 5 | |
| 4 | true | 5,4 | |
| 3 | true | 5,4,3 | |
| 2 | true | 5,4,3,2 | |
| 1 | true | 5,4,3,2,1 | |
| 0 | false | | |
| | | | First element is: 5 |
| | | | Removing first element: 5 |
| | | | First element is now: 4 |

Table D.5: Trace table for dynamic queue example

Trace Tables D.2 and D.4 are exactly the same, as are Tables D.3 and D.5. Stacks and queues can be implemented either statically or dynamically, without any apparent differences in the trace tables. The main difference between static or dynamic implementations of lists, that could play a role, is the fact that static implementations need to have the size of the list defined before run time. As such, they can reach a limit where no other elements may be inserted into the list. This is not the case with dynamic implementations of lists, since in that case, the lists may expand at run time to take any number of elements.

Section D.4.11 dealt with algorithms that used the standard library collections included in JETS. The following two **main** methods are traced, which are similar to those of section D.4.11, and make use of **ArrayList** and **LinkedList** objects respectively.

```
public static void main(String[] args) {
  ArrayList<Integer> list = new ArrayList<Integer>();

  for(int i=1; i<4; i++) {
    list.add(new Integer(i));
  }

  //Iterator example.
  Iterator<Integer> it = list.iterator();
  while(it.hasNext()) {
    System.out.print(it.next() + " ");
  }
  System.out.println();

  //List iterator example.
  ListIterator<Integer> lit = list.listIterator();
  while(lit.hasNext()) {
    Integer e = lit.next();
    lit.set(e.intValue()+10);
  }

  while(lit.hasPrevious()) {
    System.out.print(lit.previous() + " ");
  }
  System.out.println();
 }
```

| i | i<4 | list | it. hasNext() | lit. hasNext() | lit. hasPrevious() | out |
|---|---|---|---|---|---|---|
| 1 | T | [1] | | | | |
| 2 | T | [1,2] | | | | |
| 3 | T | [1,2,3] | | | | |
| 4 | F | | | | | |
| | | | T | | | 1 |
| | | | T | | | 2 |
| | | | T | | | 3 |
| | | | F | | | |
| | | [11,2,3] | | T | | |
| | | [11,12,3] | | T | | |
| | | [11,12,13] | | T | | |
| | | | | F | | |
| | | | | | T | 13 |
| | | | | | T | 12 |
| | | | | | T | 11 |
| | | | | | F | |

Table D.6: Trace table for ArrayList example

```
public static void main(String[] args) {
  LinkedList<Integer> list = new LinkedList<Integer>();

  for(int i=1; i<4; i++) {
    list.add(new Integer(i));
  }

  //Iterator example.
  Iterator<Integer> it = list.iterator();

  while(it.hasNext()) {
    System.out.print(it.next() + " ");
  }
```

```
    System.out.println();

    //List iterator example.
    ListIterator<Integer> lit = list.listIterator();

    while(lit.hasNext()) {
      Integer e = lit.next();
      lit.set(e.intValue() + 10);
    }

    while(lit.hasPrevious()) {
      System.out.print(lit.previous() + " ");
    }
    System.out.println();
  }
```

| i | i<4 | list | it. hasNext() | lit. hasNext() | lit. hasPrevious() | out |
|---|---|---|---|---|---|---|
| 1 | T | 1 | | | | |
| 2 | T | 1,2 | | | | |
| 3 | T | 1,2,3 | | | | |
| 4 | F | | | | | |
| | | | T | | | 1 |
| | | | T | | | 2 |
| | | | T | | | 3 |
| | | | F | | | |
| | | 11,2,3 | | T | | |
| | | 11,12,3 | | T | | |
| | | 11,12,13 | | T | | |
| | | | | F | | |
| | | | | | T | 13 |
| | | | | | T | 12 |
| | | | | | T | 11 |
| | | | | | F | |

**Table D.7: Trace table for LinkedList example**

Trace Tables D.6 and D.7 are exactly the same. *ArrayLists* and *LinkedLists* can be used interchangeably, without any apparent differences in the trace tables. Both may hold any number of elements.

### D.4.13 Explain the advantages of using library collections

| Exit skills. Students should be able to[1]: |
|---|
| Understand that libraries provide convenient and reliable implementations of common programming tasks. |

Anybody implementing a project of any kind does not want to "reinvent the wheel" and do extra unnecessary work. This is the case in computer programming as well. For example, if one has written a sorting algorithm for integer arrays for a previous project, one wants to avoid having to rewrite it for another project that may need it. Ideally, pre-written code, tried and tested under scrutiny, does not need to be rewritten but may be used as it is. This code may be used as a "black box" in that the programmer does not need to know the

internal workings of it, but rather only the input required, as well as the output to be delivered. For example, in a theoretical sorting algorithm for integer arrays, a programmer may call the function **sort** with an integer array as an argument in this way: **sort(integerArray)**. The programmer only needs to be aware of the existence of the **sort** function, its arguments, if any, as well as the fact that it is going to sort the elements of **integerArray** array in ascending order.

Since searching and sorting, as well as a number of other programming tasks (ex. accessing a file, sending data over a network, etc.) are common, most programming languages, Java included, come with a number of "program sets" that allow for the easy completion of these programming tasks. These "program sets" are also widely known as *libraries* or *library collections*.

As was previously seen in section D.4.11, classes (a.k.a. package members) from standard library collections may be imported using the **import** statement at the top of the file in which the classes are to be used, as such:

```
import java.util.ArrayList;
```

This import statement allows a programmer to access all the functionalities provided by the **ArrayList**. In a similar manner the following **import** statement at the top of a file allows the **sort(int[])** function described before to be used:

```
import java.util.Array.sort(int[]);
```

In order to avoid writing lots of import statements at the top of each file, one may use the asterisk (*) symbol to import all package members as follows:

```
import java.util.*;
```

All package members under the **java.util**, including **Iterator**, **ListIterator** and **ArrayList** are imported using a single statement.

Libraries provide convenient and reliable implementations of common programming tasks since:

- Code in libraries can be written and tested independently of any specific applications.
- Code in libraries may be used as a "black box" since it is known to be working. If a bug is present in the application, it exists in the application and not in the library, reducing testing and debugging time.
- Code in libraries may be used in a number of applications and programmers do not need to "reinvent the wheel".
- Code in libraries reduces the size of the application files. Programmers can call library code and avoid writing it as application code, which would make the application files long and cumbersome.

### D.4.14 Outline the features of ADT's stack, queue and binary tree

| Exit skills. Students should be able to[1]: |
| --- |
| Provide diagrams, applications and descriptions of these ADTs. For example, they should know that a binary tree can be used to efficiently store and retrieve unique keys. |

The features of ADT's stack, queue and binary tree have been discussed in detail both in Chapter 1 (Topic 5), as well as in the current chapter (Topic D). A list of these features is briefly provided below:

- **Features of Stack ADT**:
  - Stacks are made of a "chain" of data.
  - Elements can be "pushed" onto the top of the stack.
  - Elements can "popped" from the top of the stack.
  - Elements follow the LIFO approach.
  - Trying to access an element from an empty stack causes error and should be avoided.
- **Features of Queue ADT**:
  - Queues are made of a "chain" of data.
  - Elements can be "enqueued" at the back of the queue.
  - Elements can be "dequeued" from the front of the queue.
  - Elements follow the FIFO approach.
  - Trying to access an element from an empty queue causes error and should be avoided.
- **Features of Binary Tree ADT**:
  - Trees are made of nodes which have two pointers. One pointing to the left of the node to elements smaller than itself, and one pointing to the right of the node to elements greater than itself.
  - Trees assemble themselves from the root node.
  - In case of a balanced tree, the root node contains the middle value of the whole set.
  - They are naturally sorted.
  - Searching can be done in a binary manner.

### D.4.15 Explain the importance of style and naming conventions in code

| Exit skills. Students should be able to[1]: |
| --- |
| Understand that meaningful identifiers, proper indentation and adequate comments all improve the readability of code for humans and save money, time and effort in programming teams.<br>INT, AIM 5 The need to develop a common "language" to enable collaboration across international frontiers when resolving problems. |

Standards and conventions are established to make life easier. They allow people to function and collaborate at an international level without difficulty. For example, the traffic light convention dictates that a vehicle must stop if there is a red light, pass if there is a green light and be cautious at an amber light. This international convention allows people to drive in any part of the world without any problems. On the other hand, height measurements do not follow a single international convention. Some people, in some countries, use the metric system and measure height in meters and centimeters, while in other countries they measure height in feet and inches. There are a number of situations that do not follow a single international convention. This leads to difficulties in understanding and communication between people using different conventions for the same thing. Of course, a lot of these conventions go a long way back in human history, before globalization, and have been embedded so deeply in local cultures that they are difficult to change. For example, while in most of the world people drive on the right-hand side of the road, there are some countries where people drive on the left-hand side. This difference leads to drivers having difficulties switching between the two conventions, as well as increased economic costs for the car industry as a whole, since two types of cars (i.e. some with the steering wheel on the left-hand and some on the right-hand) need to be created for every car. As such, international conventions are important and allow for a common "language" that enables easier communication and collaboration between individuals across the globe.

Programming has its own conventions to allow people from around the world to easily exchange code and be able to work around the same projects. The reduction of cost, especially during the maintenance phase, is also a key reason for following coding conventions. Since most of the cost of a piece of software is taken up with the maintenance phase[7] and because the maintenance is almost always performed by someone other than the original author, code conventions are necessary to improve readability and allow for any engineer to understand previously written code quickly and fully. Code that is "clean" and follows the internationally established conventions is easier for another programmer to understand and alter, and is therefore easier and less costly to maintain.

The most common coding conventions include the following:

- Language conventions: All programming languages (except from some educational programming languages) use the English language, so as to remove a language barrier.
- Comment conventions: Comments can either be "block" (a.k.a. prologue) or "line" (a.k.a. inline) comments.
    - Block comments in Java need to be delimited by `/*` and `*/` and may span multiple lines.
    - Line comments in Java need to be delimited by `//`.
    - The best use of comments is subject to dispute, but overall they are used to summarize code.

[7] Robert L. Glass. Facts and Fallacies of Software Engineering, Addison Wesley (2003)

- Block comments can be used before every class or method to provide a general description. Line comments can be used on top or next to a line of code whose workings may not be apparent. An example is shown below:

```
/*
  The main method loops through numbers 1 to 10
  and outputs whether the number is even or odd.
*/
public static void main(String[] args) {
  int a = 0;

  //Loops through numbers 0 to 9.
  for(int i=0; i<10; i++) {

    //a is incremented in every loop.
    a = a+1;

    //Checks if a is even or odd and outputs
    //the result.
    if(a%2 == 0) {
      System.out.println(a + " is even.");
    } else {
      System.out.println(a + " is odd.");
    }
  }
}
```

- Comments should be simple and to the point. The need for a lot of comments may signify complex code that may need to be rewritten. The code on its own, using sensible naming conventions, should not need extensive commentary to be understood.

- Indent style conventions: Although in most programming languages it can be omitted, indentation of programming blocks conveys the program's structure and makes the code easier to read. There are a number of indent style conventions available[8], but all use spaces to signify that code is contained inside some other code, as can be seen by the following example:

```
public static void main(String[] args) {
  int a = 0;

  for(int i=0; i<10; i++) {
    a = a+1;
    System.out.println(a);
  }
}
```

  The code inside the **main** method is indented two white spaces to the right. The code inside the **for** loop is indented two further spaces to the right, signifying that it is both inside the **main** method and the **for** loop construct.

- Line length conventions: These refer to the maximum number of characters that may appear on a single line. Although these conventions probably originated due to

[8] Indent style. (16, July 2016). In *Wikipedia, The Free Encyclopedia*. Retrieved 15:05, July 20, 2016, from https://en.wikipedia.org/wiki/Indent_style

older technical limitations, such as various punch cards that could only consist of 80 characters in one line, they still play a role in modern programming. Screen sizes and resolutions have increased, allowing for a larger number of characters to be easily presented in one line without having to wrap around to a second. Most programming style guides define a desirable number for characters per line to be in the range of 72 to 100[9]. Having to read code that spans more than one line is hard and should be avoided when possible.

- Naming conventions: A set of rules for naming identifiers. Identifiers can be of any type of entities, such as variables, method and classes. Giving sensible names that follow specific conventions, to variables, methods and classes reduces the time and effort needed to read and understand code. It can also provide additional information for the identifier. For example, classes may always start with a capital letter, whereas object instantiations of classes may always start with a lower case letter. Naming conventions lead to consistency between individuals working on the same code, enhance clarity and reduce ambiguity. They also help avoid "naming collisions" between identifiers, leading to two identifiers with the same name. Overall, following naming conventions and using sensible names for identifiers, that describe the specific use of the identifiers, leads to better understanding of the code after a long interval of time and as such, to easier and less costly maintenance.

[9] Characters per line. (7, July 2016). In *Wikipedia, The Free Encyclopedia*. Retrieved 15:10, July 20, 2016, from https://en.wikipedia.org/wiki/Characters_per_line

**Exercise 1:** Consider the following method, `method`, where `a` is a non-negative number and `b` is a `String`.

```
public static String method(int a, String b) {
  if (a == 1) {
    return b;
  } else {
    return b + " " + method(a-1, b);
  }
}
```

1. Define recursion.
2. Trace the call to `method(5, "hi")`, showing all steps and the final output.
3. Using method `method`, construct a method with the following signature:

   `String method2(int a, String b, int c, String d)`

   that obeys the following specification:

   - if `a > c` then it first prints `String b c` times, then prints String `d`, and finally prints `String b a-c` times.
   - if `a <= c` then it prints `"a must be greater than c"`.

   For example, the a call to `method2(5, "hi", 3, "hello")` will output:

   `hi hi hi hello hi hi`

**Answer to Exercise 1:**

1. Recursion is a programming technique where a method calls itself.
2. 

```
method(5, "hi")
hi + " " + method(4, "hi")
hi + " " + hi + " " + method(3, "hi")
hi + " " + hi + " " + hi + " " + method(2, "hi")
hi + " " + hi + " " + hi + " " + hi + " " + method(1, "hi")
hi + " " + hi + " " + hi + " " + hi + " " + hi
```

   Output is: `hi hi hi hi hi`

3. 

```
public static String method2(int a, String b, int c, String d) {
  if (a > c) {
    return method(c, b) + " " + d + " " + method(a-c, b);
  } else {
    return "a must be greater than c";
  }
}
```

**Exercise 2:** Taking into account **ADTListDynamicQueue** from D.4.10, trace the following code below.

```
public static void main(String[] args) {
    ADTListDynamicQueue queue = new ADTListDynamicQueue();

    for(int i=5; i>0; i--) {
        queue.insert(i);
    }

    System.out.println("Elements in queue:");

    for(int i=0; i<queue.size(); i++) {
        System.out.println(queue.remove());
    }
  }
}
```

**Answer to Exercise 2:** A new **ADTListDynamicQueue** object is created named **queue**. The first **for** loop populates the queue, adding the elements **5**, **4**, **3**, **2** and **1**. As such, the elements in the **queue** after the first **for** loop are:

```
front - 5 4 3 2 1 - back
```

In the second for loop **i** begins from **0** to one less than the **queue** size. However, during every loop an element is removed from the **queue**. As such, after every loop, **i** increases by one, but **queue.size()** decreases by one as well.

The trace table is the following:

| i | queue | i<queue.size() | output |
|---|---|---|---|
| 5 | 5 | | |
| 4 | 5 4 | | |
| 3 | 5 4 3 | | |
| 2 | 5 4 3 2 | | |
| 1 | 5 4 3 2 1 | | |
| | 5 4 3 2 1 | | Elements in queue |
| 0 | 4 3 2 1 | 0 < 5 - true | 5 |
| 1 | 3 2 1 | 1 < 4 - true | 4 |
| 2 | 2 1 | 2 < 3 - true | 3 |
| | | 3 < 2 - false | |
| | | | |

**Exercise 3:** Construct a program that checks whether a given email address is valid. An email address is valid if it adheres to the following rules:

- The @ character is included once.
- A period is included once after the @.
- At least 2 characters are included after the period.
- There are at least 2 characters between @ and the period.
- There are at least 3 chars before the @ character.

**Answer to Exercise 3:**

```
public class email_validation
{
  public static void main(String []args)
  {
    String email = "kdimitriou@haef.gr";
    int length = email.length();
    boolean correct = false;
    int counter = 0;
    int c_counter = 0;

    for (int i=0;i<=length-1;i++) {
      if (email.charAt(i) == '@')
      {
        counter= counter + 1;
      }
    }

    if(counter==1) {
      System.out.println("The @ character is included once");
      c_counter = c_counter + 1;
    }

    int at_index = email.indexOf('@');
    String a = email.substring(at_index+1,length);
    int counter_2 = 0;

    for(int i = 0;i<a.length();i++) {
      if(a.charAt(i) == ('.')) {
        counter_2= counter_2 +1;
      }
    }

    if(counter==1) {
      System.out.println("A period is included once after the @");
      c_counter = c_counter + 1;
    }

    int index_period = a.indexOf('.');
    int length_a = a.length();
    int distance = (length_a-1) - index_period;

    if(distance>=2) {
      System.out.println("At least 2 characters are included after
the period");
      c_counter = c_counter + 1;
    }

    if(index_period>2) {
      System.out.println("There are at least 2 characters between @
and the period");
      c_counter = c_counter + 1;
    }

    if(at_index>3) {
      System.out.println("There are at least 3 chars before the @
character");
      c_counter = c_counter + 1;
    }
```

```
    if (c_counter != 5) {
      System.out.println("Email is incorrect");
    } else {
      System.out.println("Ok");
    }
  }
}
```

**Exercise 4:**

**Consider the following problem:**

Mr. Joe Doe is a high-school teacher in Athens, Greece. He teaches all school grades and is also responsible for all the computers in the school. The school started a school bookstore around ten years ago, where schools could buy and sell second hand books and magazines, to keep the prices low. Since it began as a side project and was initially small, there was no need for keeping a digital record of the books and magazines.

Currently, all reading material (books and magazines) are allocated a unique identification number and a paper card is created for each one of them that includes all their details. Whenever a student buys a book or magazine, the paper card is removed. When a student brings a book or magazine for sale, a paper card is created.

The school bookstore has grown a lot since the economic crisis hit Greece in 2010. Each year more and more students prefer to sell their books from the previous year and buy used books, for their new school year, at a lower rate. The school bookstore has grown so much and so much paper card housekeeping needs to be managed by Mr. Joe Doe alone that he is having a hard time keeping up with it. Although the school bookstore is a vital part of the school, Mr. Joe Doe cannot be offered any help, since the school is a public one and there are no funds available to recruit a full-time employee. As such, the current system, under the current circumstances is inadequate. The new application should be maintainable and there should be a way to back up all the data. The new application would need to handle all the aspects of the bookstore in a digital manner so as for Mr. Joe Doe to remove the paper cards currently in place and limit the time spent in housekeeping. The computer available for the school bookstore is an old one and running Windows XP. It has Java installed.

Develop a Java program that will correspond to the above scenario. The program should fulfil the following citeria:

1. A friendly command line interface that allows for easy use of the application.
2. A list of all the reading material in the bookstore.
3. A list of all the reading material of some specific publisher.
4. A list of all the publishers of the books or magazines in the bookstore.
5. A data entry form to allow the addition of a publisher to the bookstore.
6. The ability to remove a publisher from the bookstore.
7. A data entry form to allow the addition of a reading material to the bookstore.
8. The ability to remove a reading material from the bookstore.
9. The ability to sort the reading material in the bookstore, according to price.
10. The ability to search for a reading material, using its ID or title.

**Answer to Exercise 4:**

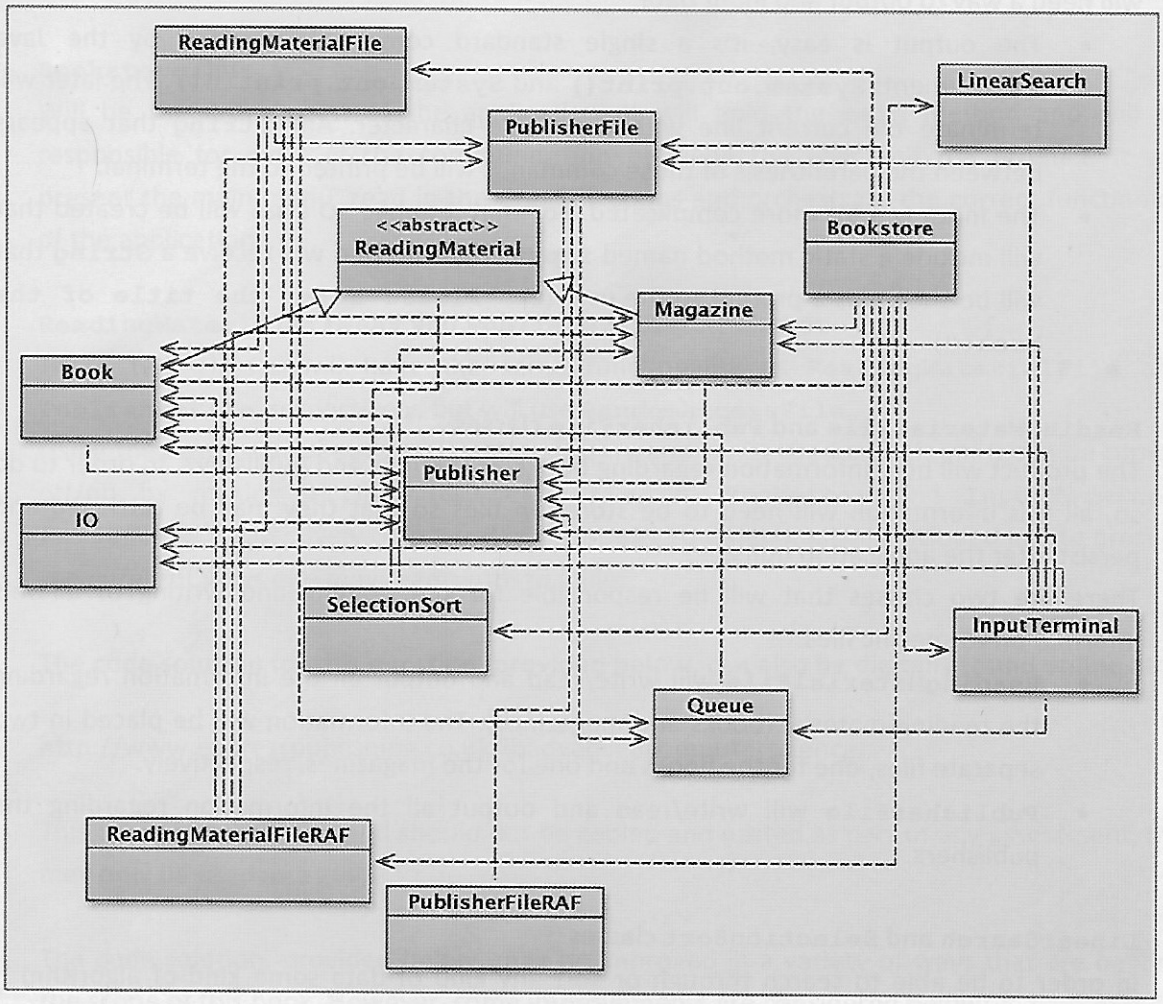



**Figure D.10: Bookstore classes**

**`ReadingMaterial`, `Book`, `Magazine` and `Publisher` classes**

The main data structures surrounding the bookstore can be the **`ReadingMaterial`**, an abstract class, that just describes the basic information related to anything that may be part of the bookstore. Since it is an abstract data structure, more concrete data structures need to be created that can be used to create actual data objects in the bookstore. These concrete data structures are the **`Book`** and the **`Magazine`**. Since they extend the capabilities of the abstract class **`ReadingMaterial`**, both will have all the variables and methods included in this class. As such, both will have an **`id`**, **`title`**, **`pages`**, **`price`** and a **`publisher`**, as well as the getter and setter methods for these variables. A book will also include chapters and a list of authors, while a magazine includes a list of additional resources it may come with (ex. a cd, recipe guide, etc.).

**`IO` class**

In order to describe a publisher, one will need more than just a primitive data variable, such as an **`int`** or a **`String`**. So, a **`Publisher`** data structure will be needed. It will include the id, name, address, telephone number and website of the publisher. Every **`Book`** or **`Magazine`** will have a single **`Publisher`**.

In order for the product to communicate with the user through a command line interface it will need a way to output and input data:

- The output is easy, it's a single standard command provided by the Java environment: `System.out.print()` and `System.out.println()`. The later will terminate the current line with a newline character. Any `String` that appears between the parenthesis of those commands will be printed to the terminal.
- The input is a bit more complicated. For that a public `IO` class will be created that will include a static method named `input`. That method will receive a `String` that will be shown as a prompt to the user (ex. `"Please enter the title of the book:"`).

### `ReadingMaterialFile` and `PublisherFile` classes

The product will hold information regarding books, magazines and publishers. In order to do so, all this information will need to be stored in files so that they may be retrieved and persist after the application quits.

There are two classes that will be responsible for the reading and writing of all this information to specific files:

- `ReadingMaterialFile` will write/read and output all the information regarding the reading material (books and magazines). The information will be placed in two separate files, one for the books and one for the magazines, respectively.
- `PublisherFile` will write/read and output all the information regarding the publishers.

### `LinearSearch` and `SelectionSort` classes

In order to be able to search through or sort any kind of data some kind of algorithm is needed:

- The `LinearSearch` class will encompass different static methods that will allow the user to search through a number of arrays holding different kinds of data, such as `Publisher`, `Book` or `Magazine` objects.
- The `SelectionSort` class will encompass different static methods that will allow the user to sort a number of arrays holding different kinds of data, such as `Book` or `Magazine` objects.

### `Queue` class

In order to be able to read the bookstore information from the files, one needs a data structure that may handle any number of elements and not just a static array that may hold a predefined number. Since the number of elements that will be read will be unknown the `Queue` class will play that role. It will provide a simple implementation of a `Queue` data structure.

### `InputTerminal` class

This will be the class that will handle the communication of the user with the product while creating new `Book`, `Magazine` or `Publisher` entries. It will be comprised of static methods

that will be called from the `Bookstore` class, where the `main` method of the application will reside.

**`Bookstore` class**

Will be the entry point of the application. It will hold the `main` method and will be responsible for most of the communication between the user and the product. It will present the main menu, read in the user selections and orchestrate the correct functioning of the application.

**`ReadingMaterialFileRAF` and `PublisherFileRAF` classes**

These two classes will have the same functionality as `ReadingMaterialFile` and `PublisherFile`, respectively, but will use `RandomAccessFile`.

Using the `RandomAccessFile` is not necessary and is not used by default. It can be turned off/on by making `withRandomAccessFile`, in `Bookstore` and `InputTerminal`, `false`/`true`, respectively. The `RandomAccessFile` functionality is included just to present the different ways one may read/write to a file.

**The code solution for this question, provided below, can also be digitally found online at:**

http://www.expresspublishing.co.uk/ibadvancedcomputerscience

This code is copyrighted and should not be copied and pasted as part of any assessment, but may only be used as a guide.

The code solution, provided below, may be improved in a variety of ways that are beyond the scope of this book. However, some improvements are considered below:

- **Additional input validation and error handling:** At the moment, the product only performs a small amount of input validations. For example, if the user does not input anything for the title of a book it will request the title again. However, there are a number of occurrences where input validation and error handling could be better. For example, when the user inputs text instead of an integer when the program is expecting the latter, or when the user inputs a decimal point number instead of an integer. The application should be able to handle such input and output a useful error message.
- **Addition of a graphical user interface (GUI) instead of command line interface (CLI):** Visual interactive elements such as buttons, lists, menus would make the use of the application a lot easier.
- **Introduction of some sort of encryption/decryption algorithm that would allow the bookstore data to be stored even more securely.**
- **Ability to make automatic backups:** At the moment backups can be performed manually by copying the three data files that have to do with books, magazines and publishers. This could be done automatically by the application to make life easier for the user. For example, it could create a copy of the files on the desktop.
- **Publisher removal if no reading material links to it.**

- **Publisher removal disallowance if there are reading materials that include it.**

Another aspect that could be improved, although it would be transparent to the users, would be the actual code of the product. There is some duplicate (identical) code in various classes (especially `ReadingMaterialFile`, `ReadingMaterialFileRAF` and `InputTerminal`) that could be combined and simplified.

```
//ReadingMaterial class

public abstract class ReadingMaterial
{
    private int id;
    private String title;
    private int pages;
    private int price;
    private Publisher publisher;

    /**
     * Constructor
     *
     * @param  id       the id of the ReadingMaterial
     * @param  title    the title of the ReadingMaterial
     * @param  pages    the number of pages of the ReadingMaterial
     * @param  price    the price of the ReadingMaterial
     * @param  publisher the publisher of the ReadingMaterial
     */
    ReadingMaterial(int id, String title, int pages, int price, Publisher publisher)
    {
        this.id = id;
        this.title = title;
        this.pages = pages;
        this.price = price;
        this.publisher = publisher;
    }

    /**
     * Accessor method
     *
     * @return        the id of the ReadingMaterial
     */
    public int getId()
    { return id; }

    /**
     * Mutator method
     *
     * @param  id    the id of the ReadingMaterial
```

```
 */
public void setId(int id)
{ this.id = id; }

/**
 * Accessor method
 *
 * @return        the title of the ReadingMaterial
 */
public String getTitle()
{ return title; }

/**
 * Mutator method
 *
 * @param  title    the title of the ReadingMaterial
 */
public void setTitle(String title)
{ this.title = title; }

/**
 * Accessor method
 *
 * @return        the number of pages of the ReadingMaterial
 */
public int getPages()
{ return pages; }

/**
 * Mutator method
 *
 * @param  pages    the number of pages of the ReadingMaterial
 */
public void setPages(int pages)
{ this.pages = pages; }

/**
 * Accessor method
 *
 * @return        the price of the Reading material
```

```
     */
    public int getPrice()
    { return price; }

     /**
      * Mutator method
      *
      * @param  price    the price of the ReadingMaterial
      */
    public void setPrice(int price)
    { this.price = price; }

     /**
      * Accessor method
      *
      * @return         the publisher of the Reading material
      */
    public Publisher getPublisher()
    { return publisher; }

     /**
      * Mutator method
      *
      * @param  publisher    the publisher of the ReadingMaterial
      */
    public void setPublisher(Publisher publisher)
    { this.publisher = publisher; }
}

//Book class

public class Book extends ReadingMaterial
{
    private int chapters;
    private String[] authors;

   /**
     * Constructor
     *
     * @param  id    the id of the book
     * @param  title    the title of the book
```

```
     * @param  pages     the number of pages of the book
     * @param  price     the price of the book
     * @param  chapters  the number of chapters in the book
     * @param  authors   a list of all the authors of the book
     * @param  publisher the publisher of the book
     */
    public Book(int id, String title, int pages, int price, int chapters, String[] authors, Publisher publisher)
    {
        //Calls the ReadingMaterial constructor.
        super(id, title, pages, price, publisher);

        this.chapters = chapters;
        this.authors = authors;
    }

    /**
     * Accessor method
     *
     * @return         the number of chapters in the Book
     */
    public int getChapters()
    { return chapters; }

    /**
     * Mutator method
     *
     * @param  chapters    the number of chapters in the Book
     */
    public void setChapters(int chapters)
    { this.chapters = chapters; }

    /**
     * Accessor method
     *
     * @return         list of all the authors of the book
     */
    public String[] getAuthors()
    { return authors; }

    /**
     * Mutator method
```

```
     *
     * @param  authors      list of all the authors of the book
     */
    public void setAuthors(String[] authors)
    { this.authors = authors; }
}

//Magazine class

public class Magazine extends ReadingMaterial
{
    private String[] additionalResources;

   /**
     * Constructor
     *
     * @param  id    the id of the book
     * @param  title    the title of the book
     * @param  pages    the number of pages of the book
     * @param  price    the price of the book
     * @param  additionalResources a list of any extra resources
     *         that may be included as part of the magazine
     *         (ex. TV guide, recipe guide, small perfume, etc.)
     * @param  publisher the publisher of the magazine
     */
    public Magazine(int id, String title, int pages, int price, String[] additionalResources, Publisher publisher)
    {
        //Calls the ReadingMaterial constructor.
        super(id, title, pages, price, publisher);

        this.additionalResources = additionalResources;
    }

    /**
     * Accessor method
     *
     * @return        list of all the resources of the magazine
     */
    public String[] getAdditionalResources()
    { return additionalResources; }
```

```
    /**
     * Mutator method
     *
     * @param  resources      set a list of all the resources of the magazine
     */
    public void setAdditionalResources(String[] resources)
    { this.additionalResources = additionalResources; }
}

//Publisher class

public class Publisher
{
    private int id;
    private String name;
    private String address;
    private int telephone;
    private String website;

   /**
     * Constructor
     *
     * @param  id          the id of the publisher
     * @param  name        the name of the publisher
     * @param  address     the address of the publisher
     * @param  telephone   the telephone of the publisher
     * @param  website     the website of the publisher
     */
    public Publisher(int id, String name, String address, int telephone, String website)
    {
        this.id = id;
        this.name = name;
        this.address = address;
        this.telephone = telephone;
        this.website = website;
    }

    /**
     * Accessor method
     *
```

```
     * @return          the id of the publisher
     */
    public int getId()
    { return id; }

    /**
     * Mutator method
     *
     * @param  id       the id of the publisher
     */
    public void setId(int id)
    { this.id = id; }

    /**
     * Accessor method
     *
     * @return          the name of the publisher
     */
    public String getName()
    { return name; }

    /**
     * Mutator method
     *
     * @param  name     the name of the publisher
     */
    public void setName(String name)
    { this.name = name; }

    /**
     * Accessor method
     *
     * @return          the address of the publisher
     */
    public String getAddress()
    { return address; }

    /**
     * Mutator method
     *
```

```
     * @param  address      the address of the publisher
     */
    public void setAddress(String address)
    { this.address = address; }

    /**
     * Accessor method
     *
     * @return        the telephone of the publisher
     */
    public int getTelephone()
    { return telephone; }

    /**
     * Mutator method
     *
     * @param  address      the telephone of the publisher
     */
    public void setTelephone(int telephone)
    { this.telephone = telephone; }

    /**
     * Accessor method
     *
     * @return        the website of the publisher
     */
    public String getWebsite()
    { return website; }

    /**
     * Mutator method
     *
     * @param  website      the website of the publisher
     */
    public void setWebsite(String website)
    { this.website = website; }
}
```

```
//IO class

public class IO
{
    static String input(String message)
    {
        String line = "";
        System.out.print(message);

        try {
            line = (new java.io.BufferedReader(new java.io.InputStreamReader(System.in))).readLine();

        } catch (Exception e) {
            line = "";
        }
        return line;
    }
}

//ReadingMaterialFile class

import java.io.PrintWriter;
import java.io.BufferedReader;
import java.io.FileReader;

public class ReadingMaterialFile
{
    static final String BOOK_FILENAME = "bookFile";
    static final String MAGAZINE_FILENAME = "magazineFile";

    //Book related methods.

    /**
     * readBookFile - reads all the book entries of the bookstore
     *
     * @return     array with all the contents of the bookFile
     * (i.e. all the books in the bookstore)
     */
    public static Book[] readBookFile()
    {
```

```
Queue books = new Queue();
Book[] arrayOfBooks = null;

try {
    FileReader theBookFile = new FileReader(BOOK_FILENAME);
    BufferedReader inputBook = new BufferedReader(theBookFile);

    String line;
    int length = 0;

    while((line = inputBook.readLine()) != null) {
        length = length + 1;

        int token = line.indexOf(":");
        int nextToken = line.indexOf(":", token+1);

        int id = Integer.parseInt(line.substring(0, token));
        String title = line.substring(token+1, nextToken);

        token = nextToken;
        nextToken = line.indexOf(":", token+1);

        int pages = Integer.parseInt(line.substring(token+1, nextToken));

        token = nextToken;
        nextToken = line.indexOf(":", token+1);

        int price = Integer.parseInt(line.substring(token+1, nextToken));

        token = nextToken;
        nextToken = line.indexOf(":", token+1);

        int chapters = Integer.parseInt(line.substring(token+1, nextToken));

        token = nextToken;
        nextToken = line.indexOf(":", token+1);

        int publisherID = Integer.parseInt(line.substring(token+1, nextToken));

        Queue authorList = new Queue();
```

```
            token = nextToken;
            nextToken = line.indexOf(":", token+1);

            while( nextToken != -1 ) {
                String author = line.substring(token+1, nextToken);
                authorList.enqueue(author);

                token = nextToken;
                nextToken = line.indexOf(":", token+1);
            }

            String[] authors = new String[authorList.size()];

            for (int i = 0; i < authors.length; i++) {
                authors[i] = (String)authorList.dequeue();
            }

            Publisher publisher = PublisherFile.readPublisherWithID(publisherID);

            Book entry = new Book(id, title, pages, price, chapters, authors, publisher);

            books.enqueue(entry);
        }

        arrayOfBooks = new Book[length];

        for(int i=0; i<length; i++) {
            Book bookFromQueue = (Book)books.dequeue();
            arrayOfBooks[i] = bookFromQueue;
        }

    } catch (Exception e) {
        //bookFile does not exist or can not be created.
        String err = e.toString();
        System.out.println("");
        System.out.println("The book file does not exist, creating it now...");
        System.out.println("");
    }
```

```
        return arrayOfBooks;
    }

    /**
     * writeToBookFile - writes a new book entry to the book file.
     *
     * @param  b   a new book entry
     */
    public static void writeToBookFile(Book b)
    {
        Book[] books = readBookFile();
        Book[] booksWithAddedBook;

        if(books != null) {

            booksWithAddedBook = new Book[books.length+1];

            for(int i=0; i < books.length; i++) {
                booksWithAddedBook[i] = books[i];
            }

            booksWithAddedBook[booksWithAddedBook.length-1] = b;
        } else {
            booksWithAddedBook = new Book[1];
            booksWithAddedBook[0] = b;
        }

        writeBooksToBookFile(booksWithAddedBook);
    }

    /**
     * writeBooksToBookFile - writes an array of books to the book file.
     *
     * @param  books   an array of books
     */
    public static void writeBooksToBookFile(Book[] books)
    {
        PrintWriter output = null;

        try {
```

```
        output = new PrintWriter(BOOK_FILENAME);
    } catch (Exception e) {
        //bookFile does not exist or can not be created.
        String err = e.toString();
        System.out.println("");
        System.out.println("The book file does not exist, creating it now...");
        System.out.println("");
    }

    if(books != null && books.length != 0) {
        for(int i=0; i < books.length; i++) {

            Book temp = books[i];

            int id = temp.getId();
            String title = temp.getTitle();
            int pages = temp.getPages();
            int price = temp.getPrice();
            int chapters = temp.getChapters();
            Publisher publisher = temp.getPublisher();

            //append them to the bookFile file.
            output.print(id + ":");
            output.print(title + ":");
            output.print(pages + ":");
            output.print(price + ":");
            output.print(chapters + ":");
            output.print(publisher.getId() + ":");

            String[] authors = temp.getAuthors();

            for(int j=0; j<authors.length; j++) {

                if(j == authors.length-1)
                    output.println(authors[j] + ":");
                else
                    output.print(authors[j] + ":");
            }
        }
```

```
        } else {
            output.print("The book file does not exist, creating it now...");
        }

        output.close();
    }

    /**
     * outputAllBooksToTerminal - outputs all the books in the bookstore to the terminal without sorting
     */
    public static void outputAllBooksToTerminal()
    {
        Book[] books = null;

        books = readBookFile();

        if(books != null) {
            if(books.length == 0) {
                System.out.println("");
                System.out.println("The bookstore is empty.");
                System.out.println("");
            }

            for(int i=0; i < books.length; i++) {

                Book temp = books[i];

                int id = temp.getId();
                String title = temp.getTitle();
                int pages = temp.getPages();
                int price = temp.getPrice();
                int chapters = temp.getChapters();
                Publisher publisher = temp.getPublisher();

                System.out.println("");
                System.out.println("==================================");
                System.out.println("ID:" + id);
                System.out.println("TITLE:" + title);
                System.out.println("PAGES:" + pages);
```

```
            System.out.println("PRICE ($):" + price);
            System.out.println("CHAPTERS:" + chapters);

            String[] authors = temp.getAuthors();

            for(int j=0; j<authors.length; j++) {

                if(!(authors[j].trim().equals("")))
                    System.out.println("AUTHOR(S):" + authors[j]);
            }

            System.out.println("PUBLISHER:" + publisher.getName());
            System.out.println("==================================");
            System.out.println("");
        }

    } else {
        System.out.println("");
        System.out.println("There are no books in the bookstore.");
        System.out.println("");
    }
}

/**
 * outputAllBooksIDsTitlesToTerminal - outputs all the books' IDs and titles
 * in the bookstore to the terminal without sorting
 *
 */
public static void outputAllBooksIDsTitlesToTerminal()
{
    Book[] books = null;

    books = readBookFile();

    if(books != null) {
        if(books.length == 0) {
            System.out.println("");
            System.out.println("The bookstore is empty.");
            System.out.println("");
        } else {
```

```
            System.out.println("");
            System.out.println("Available books.");
            System.out.println("================================");

            for(int i=0; i < books.length; i++) {

                Book temp = books[i];

                int id = temp.getId();
                String title = temp.getTitle();
                int pages = temp.getPages();
                int price = temp.getPrice();
                int chapters = temp.getChapters();
                Publisher publisher = temp.getPublisher();

                System.out.println("ID:" + id + " TITLE:" + title);

            }

            System.out.println("================================");
            System.out.println("");
        }

    } else {
        System.out.println("");
        System.out.println("There are no books in the bookstore.");
        System.out.println("");
    }
}

/**
 * outputAllBooksToTerminalOfPublisher - outputs all the books in the bookstore,
 * of a certain publisher, to the terminal without sorting
 */
public static void outputAllBooksToTerminalOfPublisher(int target)
{
    Book[] books = null;

    books = readBookFile();
```

```
if(books != null) {
    if(books.length == 0) {
        System.out.println("");
        System.out.println("The bookstore is empty.");
        System.out.println("");
    } else {

        int booksFound = 0;

        for(int i=0; i < books.length; i++) {

            Book temp = books[i];

            int id = temp.getId();
            String title = temp.getTitle();
            int pages = temp.getPages();
            int price = temp.getPrice();
            int chapters = temp.getChapters();
            Publisher publisher = temp.getPublisher();

            if(target == publisher.getId()) {

                booksFound = booksFound + 1;

                System.out.println("");
                System.out.println("================================");
                System.out.println("ID:" + id);
                System.out.println("TITLE:" + title);
                System.out.println("PAGES:" + pages);
                System.out.println("PRICE ($):" + price);

                System.out.println("CHAPTERS:" + chapters);

                String[] authors = temp.getAuthors();
                for(int j=0; j<authors.length; j++) {

                    if(!(authors[j].trim().equals("")))
                        System.out.println("AUTHOR(S):" + authors[j]);
                }
```

```
                    System.out.println("PUBLISHER:" + publisher.getName());
                    System.out.println("==================================");
                    System.out.println("");
                }

                if(booksFound == 0) {
                    System.out.println("No books found from this publisher.");
                }
            }
        }

    } else {
        System.out.println("");
        System.out.println("There are no books in the bookstore.");
        System.out.println("");
    }
}

//Magazine related methods.

/**
 * readMagazineFile - reads all the magazine entries of the bookstore
 *
 * @return     array with all the contents of the magazineFile
 * (i.e. all the magazines in the bookstore)
 */
public static Magazine[] readMagazineFile()
{
    Queue magazines = new Queue();
    Magazine[] arrayOfMagazines = null;

    try {
        FileReader theMagazineFile = new FileReader(MAGAZINE_FILENAME);
        BufferedReader inputMagazine = new BufferedReader(theMagazineFile);

        String line;
        int length = 0;
```

```
while((line = inputMagazine.readLine()) != null) {
    length = length + 1;

    int token = line.indexOf(":");
    int nextToken = line.indexOf(":", token+1);

    int id = Integer.parseInt(line.substring(0, token));
    String title = line.substring(token+1, nextToken);

    token = nextToken;
    nextToken = line.indexOf(":", token+1);

    int pages = Integer.parseInt(line.substring(token+1, nextToken));

    token = nextToken;
    nextToken = line.indexOf(":", token+1);

    int price = Integer.parseInt(line.substring(token+1, nextToken));

    token = nextToken;
    nextToken = line.indexOf(":", token+1);

    int publisherID = Integer.parseInt(line.substring(token+1, nextToken));

    Queue resourcesList = new Queue();

    token = nextToken;
    nextToken = line.indexOf(":", token+1);

    while( nextToken != -1 ) {
        String resource = line.substring(token+1, nextToken);
        resourcesList.enqueue(resource);

        token = nextToken;
        nextToken = line.indexOf(":", token+1);
    }

    String[] resources = new String[resourcesList.size()];

    for (int i = 0; i < resources.length; i++) {
```

```
                resources[i] = (String)resourcesList.dequeue();
            }

            Publisher publisher = PublisherFile.readPublisherWithID(publisherID);

            Magazine entry = new Magazine(id, title, pages, price, resources, publisher);

            magazines.enqueue(entry);
        }

        arrayOfMagazines = new Magazine[length];

        for(int i=0; i<length; i++) {
            Magazine magazineFromQueue = (Magazine)magazines.dequeue();
            arrayOfMagazines[i] = magazineFromQueue;
        }

    } catch (Exception e) {
        //magazineFile does not exist or can not be created.
        String err = e.toString();
        System.out.println("");
        System.out.println("The magazine file does not exist, creating it now...");
        System.out.println("");
    }

    return arrayOfMagazines;
}

/**
 * writeToMagazineFile - writes a new magazine entry to the magazine file.
 *
 * @param  m   a new magazine entry
 */
public static void writeToMagazineFile(Magazine m)
{
    Magazine[] magazines = readMagazineFile();
    Magazine[] magazinesWithAddedMagazine;

    if(magazines != null) {
```

```
            magazinesWithAddedMagazine = new Magazine[magazines.length+1];

            for(int i=0; i < magazines.length; i++) {
                magazinesWithAddedMagazine[i] = magazines[i];
            }

            magazinesWithAddedMagazine[magazinesWithAddedMagazine.length-1] = m;
        } else {
            magazinesWithAddedMagazine = new Magazine[1];
            magazinesWithAddedMagazine[0] = m;
        }

        writeMagazinesToMagazineFile(magazinesWithAddedMagazine);
    }

    /**
     * writeMagazinesToMagazineFile - writes an array of magazines to the magazine file.
     *
     * @param  magazines   an array of magazines
     */
    public static void writeMagazinesToMagazineFile(Magazine[] magazines)
    {
        PrintWriter output = null;

        try {
            output = new PrintWriter(MAGAZINE_FILENAME);
        } catch (Exception e) {
            //magazineFile does not exist or can not be created.
            String err = e.toString();
            System.out.println("");
            System.out.println("The magazine file does not exist, creating it now...");
            System.out.println("");
        }

        if(magazines != null && magazines.length != 0) {
            for(int i=0; i < magazines.length; i++) {

                Magazine temp = magazines[i];

                int id = temp.getId();
```

```
                String title = temp.getTitle();
                int pages = temp.getPages();
                int price = temp.getPrice();

                Publisher publisher = temp.getPublisher();

                //append them to the magazineFile file.
                output.print(id + ":");
                output.print(title + ":");
                output.print(pages + ":");
                output.print(price + ":");
                output.print(publisher.getId() + ":");

                String[] resources = temp.getAdditionalResources();

                for(int j=0; j<resources.length; j++) {

                    if(j == resources.length-1)
                        output.println(resources[j] + ":");
                    else
                        output.print(resources[j] + ":");
                }
            }

        } else {
            output.print("The magazine file does not exist, creating it now...");
        }

        output.close();
    }

    /**
     * outputAllMagazinesToTerminal - outputs all the magazines in the bookstore
     * to the terminal without sorting
     */
    public static void outputAllMagazinesToTerminal()
    {
        Magazine[] magazines = null;
        magazines = readMagazineFile();

        if(magazines != null) {
```

```
        if(magazines.length == 0) {
            System.out.println("");
            System.out.println("The bookstore is empty.");
            System.out.println("");
        }

        for(int i=0; i < magazines.length; i++) {

            Magazine temp = magazines[i];

            int id = temp.getId();
            String title = temp.getTitle();
            int pages = temp.getPages();
            int price = temp.getPrice();

            Publisher publisher = temp.getPublisher();

            System.out.println("");
            System.out.println("==================================");
            System.out.println("ID:" + id);
            System.out.println("TITLE:" + title);
            System.out.println("PAGES:" + pages);
            System.out.println("PRICE ($):" + price);

            String[] resources = temp.getAdditionalResources();

            for(int j=0; j<resources.length; j++) {

                if(!(resources[j].trim().equals("")))
                    System.out.println("RESOURCE(S):" + resources[j]);
            }

            System.out.println("PUBLISHER:" + publisher.getName());
            System.out.println("==================================");
            System.out.println("");
        }

    } else {
        System.out.println("");
        System.out.println("There are no magazines in the bookstore.");
        System.out.println("");
```

```
    }
}

/**
 * outputAllMagazinesIDsTitlesToTerminal - outputs all the magazines' IDs and titles
 * in the bookstore to the terminal without sorting
 */
public static void outputAllMagazinesIDsTitlesToTerminal()
{
    Magazine[] magazines = null;
    magazines = readMagazineFile();

    if(magazines != null) {
        if(magazines.length == 0) {
            System.out.println("");
            System.out.println("The bookstore is empty.");
            System.out.println("");
        } else {

            System.out.println("");
            System.out.println("Available magazines.");
            System.out.println("================================");

            for(int i=0; i < magazines.length; i++) {

                Magazine temp = magazines[i];

                int id = temp.getId();
                String title = temp.getTitle();
                int pages = temp.getPages();
                int price = temp.getPrice();

                Publisher publisher = temp.getPublisher();

                System.out.println("ID:" + id + " TITLE:" + title);

            }

            System.out.println("================================");
            System.out.println("");
        }
```

```
        } else {
            System.out.println("");
            System.out.println("There are no magazines in the bookstore.");
            System.out.println("");
        }
    }

    /**
     * outputAllMagazinesToTerminalOfPublisher - outputs all the magazines in the bookstore,
     * of a certain publisher, to the terminal without sorting
     */
    public static void outputAllMagazinesToTerminalOfPublisher(int target)
    {
        Magazine[] magazines = null;
        magazines = readMagazineFile();

        if(magazines != null) {
            if(magazines.length == 0) {
                System.out.println("");
                System.out.println("The bookstore is empty.");
                System.out.println("");
            } else {

                int magazinesFound = 0;

                for(int i=0; i < magazines.length; i++) {

                    Magazine temp = magazines[i];

                    int id = temp.getId();
                    String title = temp.getTitle();
                    int pages = temp.getPages();
                    int price = temp.getPrice();

                    Publisher publisher = temp.getPublisher();

                    if(target == publisher.getId()) {

                        magazinesFound = magazinesFound + 1;
```

```
                        System.out.println("");
                        System.out.println("==================================");
                        System.out.println("ID:" + id);
                        System.out.println("TITLE:" + title);
                        System.out.println("PAGES:" + pages);
                        System.out.println("PRICE ($):" + price);

                        String[] resources = temp.getAdditionalResources();
                        for(int j=0; j<resources.length; j++) {

                            if(!(resources[j].trim().equals("")))
                                System.out.println("RESOURCE(S):" + resources[j]);
                        }

                        System.out.println("PUBLISHER:" + publisher.getName());
                        System.out.println("==================================");
                        System.out.println("");
                    }

                    if(magazinesFound == 0) {
                        System.out.println("No magazines found from this publisher.");
                    }
                }
            }

        } else {
            System.out.println("");
            System.out.println("There are no magazines in the bookstore.");
            System.out.println("");
        }
    }
}

//PublisherFile class

import java.io.PrintWriter;
import java.io.BufferedReader;
import java.io.FileReader;

public class PublisherFile
{
```

```
static final String PUBLISHER_FILENAME = "publisherFile";

/**
 * readPublisherFile - reads all the publisher entries
 *
 * @return     array with all the contents of the publisherFile
 */
public static Publisher[] readPublisherFile()
{
    Queue publishers = new Queue();
    Publisher[] arrayOfPublishers = null;

    try {
        FileReader thePublisherFile = new FileReader(PUBLISHER_FILENAME);
        BufferedReader inputPublisher = new BufferedReader(thePublisherFile);

        String line;
        int length = 0;

        while((line = inputPublisher.readLine()) != null) {
            length = length + 1;

            int token = line.indexOf(":");
            int nextToken = line.indexOf(":", token+1);

            int id = Integer.parseInt(line.substring(0, token));

            String name = line.substring(token+1, nextToken);

            token = nextToken;
            nextToken = line.indexOf(":", token+1);

            String address = line.substring(token+1, nextToken);

            token = nextToken;
            nextToken = line.indexOf(":", token+1);

            int telephone = Integer.parseInt(line.substring(token+1, nextToken));

            token = nextToken;
```

```
            nextToken = line.indexOf(":", token+1);

            String website = line.substring(token+1, nextToken);

            Publisher entry = new Publisher(id, name, address, telephone, website);

            publishers.enqueue(entry);
        }

        arrayOfPublishers = new Publisher[length];

        for(int i=0; i<length; i++) {

            Publisher publisherFromQueue = (Publisher)publishers.dequeue();

            int id = publisherFromQueue.getId();
            String name = publisherFromQueue.getName();
            String address = publisherFromQueue.getAddress();
            int telephone = publisherFromQueue.getTelephone();
            String website = publisherFromQueue.getWebsite();

            arrayOfPublishers[i] = new Publisher(id, name, address, telephone, website);

        }

    } catch (Exception e) {
        //publisherFile does not exist or cannot be created.
        String err = e.toString();
        System.out.println("");
    }

    return arrayOfPublishers;
}

/**
 * writeToPublisherFile - writes a new publisher entry to the publisher file.
 *
 * @param  p    a new publisher entry
 */
public static void writeToPublisherFile(Publisher p)
```

```
{
    Publisher[] publishers = readPublisherFile();
    Publisher[] publishersWithAddedPublisher;

    if(publishers != null) {

        publishersWithAddedPublisher = new Publisher[publishers.length+1];

        for(int i=0; i < publishers.length; i++) {
            publishersWithAddedPublisher[i] = publishers[i];
        }

        publishersWithAddedPublisher[publishersWithAddedPublisher.length-1] = p;
    } else {
        publishersWithAddedPublisher = new Publisher[1];
        publishersWithAddedPublisher[0] = p;
    }

    writePublishersToPublisherFile(publishersWithAddedPublisher);
}
```

```
/**
 * writePublishersToPublisherFile - writes an array of publishers to the publisher file.
 *
 * @param  publishers   an array of publishers
 */
public static void writePublishersToPublisherFile(Publisher[] publishers)
{
    PrintWriter output = null;

    try {
        output = new PrintWriter(PUBLISHER_FILENAME);
    } catch (Exception e) {
        //publisherFile does not exist or can not be created.
        String err = e.toString();
        System.out.println("");
        System.out.println("The publisher file cannot be created.");
        System.out.println("");
    }

    if(publishers != null && publishers.length != 0) {
        for(int i=0; i < publishers.length; i++) {

            Publisher temp = publishers[i];

            int id = temp.getId();
            String name = temp.getName();
            String address = temp.getAddress();
            int telephone = temp.getTelephone();
            String website = temp.getWebsite();

            //append them to the publisherFile file.
            output.print(id + ":");
            output.print(name + ":");
            output.print(address + ":");
            output.print(telephone + ":");
            output.println(website + ":");
        }

    } else {
        output.print("");
```

```
        }

        output.close();
    }

    /**
     * readPublisherWithID - returns the publisher with a specific id.
     *
     * @param  id   the publisher id
     */
    public static Publisher readPublisherWithID(int id)
    {
        Publisher[] publishers = readPublisherFile();
        Publisher publisher;

        for(int i=0; i<publishers.length; i++) {
            if(publishers[i].getId() == id) {
                return publisher = publishers[i];
            }
        }

        return null;
    }

    /**
     * outputAllPublishersToTerminal - outputs all the publishers to the terminal without sorting
     */
    public static void outputAllPublishersToTerminal()
    {
        Publisher[] publishers = null;

        publishers = readPublisherFile();

        if(publishers != null) {
            if(publishers.length == 0) {
                System.out.println("");
                System.out.println("The are no publishers.");
                System.out.println("");
            }
```

```
        for(int i=0; i < publishers.length; i++) {

            Publisher temp = publishers[i];

            int id = temp.getId();
            String name = temp.getName();
            String address = temp.getAddress();
            int telephone = temp.getTelephone();
            String website = temp.getWebsite();

            System.out.println("");
            System.out.println("==================================");
            System.out.println("ID:" + id);
            System.out.println("Name:" + name);
            System.out.println("Address:" + address);
            System.out.println("Telephone:" + telephone);
            System.out.println("Website:" + website);
            System.out.println("==================================");
            System.out.println("");
        }

    } else {
        System.out.println("");
        System.out.println("There are no publishers.");
        System.out.println("");
    }
}

/**
 * outputAllPublishersIDsNamesToTerminal - outputs all the publishers' ID and names
 * to the terminal without sorting
 *
 */
public static void outputAllPublishersIDsNamesToTerminal()
{
    Publisher[] publishers = null;

    publishers = readPublisherFile();

    if(publishers != null) {
```

```
            if(publishers.length == 0) {
                System.out.println("");
                System.out.println("The are no publishers.");
                System.out.println("");
            } else {

                System.out.println("");
                System.out.println("Available publishers:");
                System.out.println("================================");

                for(int i=0; i < publishers.length; i++) {

                    Publisher temp = publishers[i];

                    int id = temp.getId();
                    String name = temp.getName();
                    String address = temp.getAddress();
                    int telephone = temp.getTelephone();
                    String website = temp.getWebsite();

                    System.out.println("ID:" + id + ", NAME:" + name);
                }
                System.out.println("================================");
                System.out.println("");
            }

        } else {
            System.out.println("");
            System.out.println("There are no publishers.");
            System.out.println("");
        }
    }
}

//LinearSearch class

public class LinearSearch
{
    /**
     * linearSearch - searches an array of Publisher records for a specific publisher.
```

```
 *
 * @param  target          the name to be found
 * @param  publishers      the array of Publisher records to be searched
 * @return     -1 if element not found, else the index where our element resides in the array.
 */
public static int linearSearch(String target, Publisher[] publishers)
{
    int size = publishers.length;
    boolean found = false;
    int place = 0;
    while (place < size && !found)
    {
        String toFind = publishers[place].getName();
        String toFindTrim = toFind.trim();
        if (target.equals(toFindTrim))
        { found = true; }
        else
        { place = place + 1; }
    }

    if (found)
    { return place; }
    else
    { return -1; }
}

/**
 * linearSearch - searches an array of Publisher records for a specific publisher.
 *
 * @param  target          the id to be found
 * @param  publishers      the array of Publisher records to be searched
 * @return     -1 if element not found, else the index where our element resides in the array.
 */
public static int linearSearch(int target, Publisher[] publishers)
{
    int size = publishers.length;
    boolean found = false;
    int place = 0;
    while (place < size && !found)
    {
        int toFind = publishers[place].getId();
```

```
            if (target == toFind)
            { found = true; }

            if(!found)
            { place = place + 1; }
        }

        if (found)
        { return place; }
        else
        { return -1; }
    }

    /**
     * linearSearch - searches an array of books for a specific title.
     *
     * @param  target    the title of the book to be found
     * @param  books     the array of books to be searched
     * @return     -1 if element not found, else the index where our element resides in the array.
     */
    public static int linearSearch(String target, Book[] books)
    {
        int size = books.length;
        boolean found = false;
        int place = 0;
        while (place < size && !found)
        {
            String toFind = books[place].getTitle();
            String toFindTrim = toFind.trim();
            if (target.equals(toFindTrim))
            { found = true; }
            else
            { place = place + 1; }
        }

        if (found)
        { return place; }
        else
        { return -1; }
    }
```

```
/**
 * linearSearch - searches an array of books for a specific ID.
 *
 * @param  target    the ID of the book to be found
 * @param  books     the array of books to be searched
 * @return     -1 if element not found, else the index where our element resides in the array.
 */
public static int linearSearch(int target, Book[] books)
{
    int size = books.length;
    boolean found = false;
    int place = 0;
    while (place < size && !found)
    {
        int toFind = books[place].getId();

        if (target == toFind)
        { found = true; }
        else
        { place = place + 1; }
    }

    if (found)
    { return place; }
    else
    { return -1; }
}

/**
 * linearSearch - searches an array of books for a specific ID.
 *
 * @param  target    the title of the magazine to be found
 * @param  magazines     the array of magazines to be searched
 * @return     -1 if element not found, else the index where our element resides in the array.
 */
public static int linearSearch(String target, Magazine[] magazines)
{
    int size = magazines.length;
    boolean found = false;
    int place = 0;
```

```
    while (place < size && !found)
    {
        String toFind = magazines[place].getTitle();
        String toFindTrim = toFind.trim();
        if (target.equals(toFindTrim))
        { found = true; }
        else
        { place = place + 1; }
    }

    if (found)
    { return place; }
    else
    { return -1; }
}

/**
 * linearSearch - searches an array of books for a specific ID.
 *
 * @param  target    the ID of the magazine to be found
 * @param  magazines     the array of magazines to be searched
 * @return     -1 if element not found, else the index where our element resides in the array.
 */
public static int linearSearch(int target, Magazine[] magazines)
{
    int size = magazines.length;
    boolean found = false;
    int place = 0;
    while (place < size && !found)
    {
        int toFind = magazines[place].getId();

        if (target == toFind)
        { found = true; }
        else
        { place = place + 1; }
    }

    if (found)
    { return place; }
    else
```

```
        { return -1; }
    }
}

//Selection Sort class

public class SelectionSort
{
    /**
     * selectionSort - sorts the books in the bookstore by price
     *
     * @param books     an array containing the books to be sorted
     * @param ascending a boolean value indicating whether sorting should be ascending or descending
     */
    public static Book[] selectionSort(Book[] books, boolean ascending)
    {
        int first, current, least;
        Book temp;

        for(first = 0; first < books.length; first = first + 1)
        {
            least = first;
            for(current = first+1; current < books.length; current = current + 1)
            {
                if(ascending) {
                    if (books[current].getPrice() > books[least].getPrice())
                    { least = current; }
                } else {
                    if (books[current].getPrice() < books[least].getPrice())
                    { least = current; }
                }
            }
            temp = books[least];
            books[least] = books[first];
            books[first] = temp;
        }

        return books;
    }
```

```
    /**
     * selectionSort - sorts the magazines in the bookstore by price
     *
     * @param magazines an array containing the magazines to be sorted
     * @param ascending a boolean value indicating whether sorting should be ascending or descending
     */
    public static Magazine[] selectionSort(Magazine[] magazines, boolean ascending)
    {
        int first, current, least;
        Magazine temp;

        for(first = 0; first < magazines.length; first = first + 1)
        {
            least = first;
            for(current = first+1; current < magazines.length; current = current + 1)
            {
                if(ascending) {
                    if (magazines[current].getPrice() > magazines[least].getPrice())
                    { least = current; }
                } else {
                    if (magazines[current].getPrice() < magazines[least].getPrice())
                    { least = current; }
                }
            }
            temp = magazines[least];
            magazines[least] = magazines[first];
            magazines[first] = temp;
        }

        return magazines;
    }
}

//Queue class

import java.util.ArrayList;

public class Queue {
    private ArrayList<Object> queue;

    public Queue() {
```

```
        queue = new ArrayList<Object>();
    }

    public void enqueue(Object o) {
        queue.add(o);
    }

    public Object dequeue() {
        if (queue.isEmpty()) {
            System.out.println("Error: queue is empty.");
            return null;
        } else {

            Object number = (Object)queue.get(0);

            for (int i = 0; i < queue.size() - 1; i++) {
                queue.set(i, queue.get(i+1));
            }
            queue.remove(queue.size() - 1);

            return number;
        }
    }

    public int size() {
        return queue.size();
    }
}

// InputTerminal class

public class InputTerminal
{
     /**
     * WARNING: in order for the program to fuction correctly
     * with the withRandomAccessFile = true the ReadingMaterialFileRAF
     * class must be completed. This class is completed as part of the
     * exercises of this Chapter in the book.
     *
     * Furthermore, a PublisherFileRAF will also need to be
```

```
 * completed in order for the publishers to be read and written
 * to a binary file. This class is, once again, completed as
 * part of the exercises of this Chapter in the book.
 *
 * As such, make withRandomAccessFile = true only when the
 * exercises have been completed successfuly.
 */
static final boolean withRandomAccessFile = false;

 /**
 * publisherEntry method
 *
 * Responsible for getting user input for a new publisher entry in the bookstore.
 */
public static Publisher publisherEntry()
{
    String name, address, website;
    int id, telephone;

    System.out.println("");
    System.out.println("");
    System.out.println("====================");
    System.out.println("New bookstore entry");
    System.out.println("====================");
    System.out.println("");
    System.out.println("");
    System.out.println("");
    System.out.println("Publisher details");
    System.out.println("-----------------");
    System.out.println("");

    id = Integer.valueOf(IO.input("id: ").trim()).intValue();
    while(id < 0 || id == 0) {
        System.out.println("Error: The id you entered was not a proper id number");
        id = Integer.valueOf(IO.input("id: ").trim()).intValue();
    }

    name = IO.input("name: ");
    while(name.trim().length() == 0) {
        System.out.println("Error: The name you entered was improper");
        name = IO.input("name: ");
    }
```

```
        address = IO.input("address: ");
        while(address.trim().length() == 0) {
            System.out.println("Error: The address you entered was improper");
            address = IO.input("address: ");
        }

        telephone = Integer.valueOf(IO.input("telephone number: ").trim()).intValue();
        while(id < 0 || id == 0) {
            System.out.println("Error: The telephone number you entered was not a proper telephone number");
            telephone = Integer.valueOf(IO.input("telephone number: ").trim()).intValue();
        }

        website = IO.input("website: ");
        while(website.trim().length() == 0) {
            System.out.println("Error: The website you entered was improper");
            address = IO.input("website: ");
        }

        Publisher p = new Publisher(id, name, address, telephone, website);

        return p;
    }

    /**
     * getIOMethod method
     *
     * @return      whether we are using RandomAccessFile or Sequential access for input and output.
     */
    public static boolean getIOMethod()
    {
        return withRandomAccessFile;
    }

    /**
     * bookEntry method
     *
     * Responsible for getting user input for a new book entry in the bookstore.
     */
    public static Book bookEntry()
    {
        String title;
```

```
int id, pages, price, chapters;
Publisher publisher;

System.out.println("");
System.out.println("");
System.out.println("====================");
System.out.println("New bookstore entry");
System.out.println("====================");
System.out.println("");
System.out.println("");
System.out.println("");
System.out.println("Book details");
System.out.println("------------");
System.out.println("");

id = Integer.valueOf(IO.input("id: ").trim()).intValue();
while(id < 0 || id == 0) {
    System.out.println("Error: The id you entered was not a proper id number");
    id = Integer.valueOf(IO.input("id: ").trim()).intValue();
}

title = IO.input("title: ");
while(title.trim().length() == 0 || title.trim().length() > 25) {
    System.out.println("Error: The title you entered was empty or has more than 25 characters");
    title = IO.input("title: ");
}

pages = Integer.valueOf(IO.input("pages: ").trim()).intValue();
while(pages < 0 || pages == 0) {
    System.out.println("Error: The pages you entered was not a positive number");
    pages = Integer.valueOf(IO.input("pages: ").trim()).intValue();
}

price = Integer.valueOf(IO.input("price ($): ").trim()).intValue();
while(price < 0 || price == 0) {
    System.out.println("Error: The price you entered was not a positive number");
    price = Integer.valueOf(IO.input("price ($): ").trim()).intValue();
}

chapters = Integer.valueOf(IO.input("chapters: ").trim()).intValue();
while(chapters < 0 || chapters == 0) {
```

```
            System.out.println("Error: The chapters you entered was not a positive number");
            chapters = Integer.valueOf(IO.input("chapters: ").trim()).intValue();
        }

        System.out.println("");

        if(!getIOMethod()) {
                //sequentially
                PublisherFile.outputAllPublishersIDsNamesToTerminal();
                int publisherId = Integer.valueOf(IO.input("publisher id: ").trim()).intValue();
                publisher = PublisherFile.readPublisherWithID(publisherId);

                while(price < 0 || price == 0 || publisher == null) {
                    System.out.println("Error: The id you entered was not a proper id number");
                    publisherId = Integer.valueOf(IO.input("publisher id: ").trim()).intValue();
                    publisher = PublisherFile.readPublisherWithID(publisherId);
                }

        } else {
                //random access
                PublisherFileRAF.outputAllPublishersIDsNamesToTerminal();
                int publisherId = Integer.valueOf(IO.input("publisher id: ").trim()).intValue();
                publisher = PublisherFileRAF.readPublisherWithID(publisherId);

                while(price < 0 || price == 0 || publisher == null) {
                    System.out.println("Error: The id you entered was not a proper id number");
                    publisherId = Integer.valueOf(IO.input("publisher id: ").trim()).intValue();
                    publisher = PublisherFileRAF.readPublisherWithID(publisherId);
                }
        }

        String[] authorArray = authorEntry();
        Book b = new Book(id, title, pages, price, chapters, authorArray, publisher);

        return b;
    }

    /**
     * authorEntry method
     *
     * Responsible for getting user input for the authors of a new book entry in the bookstore.
     */
```

```
public static String[] authorEntry()
{
    Queue authors = new Queue();
    String author;

    char moreAuthors = 'y';
    int authorNum = 1;

    while(moreAuthors == 'y' || moreAuthors == 'Y') {
        author = IO.input("author " + authorNum + ": ");

        while(author.trim().length() == 0 || author.trim().length() > 25) {
            System.out.println("Error: The author name you entered was empty or has more than 25 characters");
            author = IO.input("author " + authorNum + ": ");
        }

        authors.enqueue(author);
        authorNum = authorNum+1;

        //a book can not have more than 4 authors.
        if(authorNum > 4) moreAuthors = 'n';
        else {
            do {
                moreAuthors = IO.input("        is there another author, Yes (y) or No (n)? ").charAt(0);
            } while(moreAuthors != 'y' && moreAuthors != 'Y' && moreAuthors != 'n' && moreAuthors != 'N');

        }
    }

    String[] authorArray = new String[authors.size()];

    for (int i = 0; i < authorArray.length; i++) {
        authorArray[i] = (String)authors.dequeue();
    }

    return authorArray;
}

/**
 * magazineEntry method
 *
```

```
 * Responsible for getting user input for a new magazine entry in the bookstore.
 */
public static Magazine magazineEntry()
{
    String title;
    int id, pages, price;
    Publisher publisher;

    System.out.println("");
    System.out.println("");
    System.out.println("====================");
    System.out.println("New bookstore entry");
    System.out.println("====================");
    System.out.println("");
    System.out.println("");
    System.out.println("");
    System.out.println("Magazine details");
    System.out.println("----------------");
    System.out.println("");

    id = Integer.valueOf(IO.input("id: ").trim()).intValue();
    while(id < 0 || id == 0) {
        System.out.println("Error: The id you entered was not a proper id number");
        id = Integer.valueOf(IO.input("id: ").trim()).intValue();
    }

    title = IO.input("title: ");
    while(title.trim().length() == 0 || title.trim().length() > 25) {
        System.out.println("Error: The title you entered was empty or has more than 25 characters");
        title = IO.input("title: ");
    }

    pages = Integer.valueOf(IO.input("pages: ").trim()).intValue();
    while(pages < 0 || pages == 0) {
        System.out.println("Error: The pages you entered was not a positive number");
        pages = Integer.valueOf(IO.input("pages: ").trim()).intValue();
    }

    price = Integer.valueOf(IO.input("price ($): ").trim()).intValue();
    while(price < 0 || price == 0) {
        System.out.println("Error: The price you entered was not a positive number");
        price = Integer.valueOf(IO.input("price ($): ").trim()).intValue();
```

```
        }

        System.out.println("");

        if(!getIOMethod()) {
                //sequentially
                PublisherFile.outputAllPublishersIDsNamesToTerminal();
                int publisherId = Integer.valueOf(IO.input("publisher id: ").trim()).intValue();
                publisher = PublisherFile.readPublisherWithID(publisherId);

                while(price < 0 || price == 0 || publisher == null) {
                    System.out.println("Error: The id you entered was not a proper id number");
                    publisherId = Integer.valueOf(IO.input("publisher id: ").trim()).intValue();
                    publisher = PublisherFile.readPublisherWithID(publisherId);
                }

        } else {
                //random access
                PublisherFileRAF.outputAllPublishersIDsNamesToTerminal();
                int publisherId = Integer.valueOf(IO.input("publisher id: ").trim()).intValue();
                publisher = PublisherFileRAF.readPublisherWithID(publisherId);

                while(price < 0 || price == 0 || publisher == null) {
                    System.out.println("Error: The id you entered was not a proper id number");
                    publisherId = Integer.valueOf(IO.input("publisher id: ").trim()).intValue();
                    publisher = PublisherFileRAF.readPublisherWithID(publisherId);
                }
        }

        String[] resources = resourceEntry();
        Magazine m = new Magazine(id, title, pages, price, resources, publisher);

        return m;
    }

    /**
     * resourceEntry method
     *
     * Responsible for getting user input for the resouces of a new magazine entry in the bookstore.
     */
    public static String[] resourceEntry()
```

```
    {
        Queue resources = new Queue();
        String resource;

        char moreResources = 'y';
        int resourceNum = 1;

        while(moreResources == 'y' || moreResources == 'Y') {
            resource = IO.input("resource " + resourceNum + ": ");

            while(resource.trim().length() == 0 || resource.trim().length() > 25) {
                System.out.println("Error: The resource entered was empty or has more than 25 characters");
                resource = IO.input("resource " + resourceNum + ": ");
            }

            resources.enqueue(resource);
            resourceNum = resourceNum+1;

            //a magazine can not have more than 4 resources.
            if(resourceNum > 4) moreResources = 'n';
            else {
                do {
                    moreResources = IO.input("      is there another resource, Yes (y) or No (n)? ").charAt(0);
                } while(moreResources != 'y' && moreResources != 'Y' && moreResources != 'n' && moreResources != 'N');

            }
        }

        String[] resourceArray = new String[resources.size()];

        for (int i = 0; i < resourceArray.length; i++) {
            resourceArray[i] = (String)resources.dequeue();
        }

        return resourceArray;
    }
}

 //Bookstore class

 public class Bookstore
```

```
{
    /**
     * WARNING: in order for the program to fuction correctly
     * with the withRandomAccessFile = true the ReadingMaterialFileRAF
     * class must be completed. This class is completed as part of the
     * exercises of this Chapter in the book.
     *
     * Furthermore, a PublisherFileRAF will also need to be
     * completed in order for the publishers to be read and written
     * to a binary file. This class is, once again, completed as
     * part of the exercises of this Chapter in the book.
     *
     * As such, make withRandomAccessFile = true only when the
     * exercises have been completed successfuly.
     */
    static final boolean withRandomAccessFile = false;

    /**
     * Main method
     */
    public static void main(String[] args)
    {
        new Bookstore();
    }

    /**
     * Constructor for objects of class Bookstore
     */
    public Bookstore()
    {
        showMenu();
    }

    /**
     * showMenu method - displays the start menu in the Terminal.
     *
     */
    public void showMenu()
    {
         //*** BlueJ specific code
        //Used to clear the terminal
```

```
//should be commented out if used in other environments
System.out.println('\f');
//*** BlueJ specific code

System.out.println("========================");
System.out.println("Welcome to the Bookstore");
System.out.println("========================");
System.out.println("");
System.out.println("Select one of the following actions:");
System.out.println("a) List all the publishers");
System.out.println("b) Add a publisher");
System.out.println("c) Remove a publisher");
System.out.println("d) List all the reading materials in the bookstore");
System.out.println("e) List all the reading materials of specific publisher");
System.out.println("f) Add a reading material to the bookstore");
System.out.println("g) Remove a reading material from the bookstore");
System.out.println("h) Sort reading materials by price (ascending or descending)");
System.out.println("i) Search for reading material (ID or title)");
System.out.println("j) Exit");
System.out.println("");
System.out.println("");

char selection;

do {
    selection = IO.input("Which action do you want to perform? ").charAt(0);
} while(selection != 'a' && selection != 'b' && selection != 'c' && selection != 'd'
&& selection != 'e' && selection != 'f' && selection != 'g' && selection != 'h'
&& selection != 'i' && selection != 'j');

switch (selection) {
    case 'a' : listAllPublishers();
    break;
    case 'b' : addPublisher();
    break;
    case 'c' : removePublisher();
    break;
    case 'd' : listAllReadingMaterial();
    break;
    case 'e' : listReadingMaterialByPublisher();
    break;
```

```
            case 'f' : addReadingMaterial();
            break;
            case 'g' : removeReadingMaterial();
            break;
            case 'h' : sortReadingMaterialByPrice();
            break;
            case 'i' : findReadingMaterial();
            break;
            case 'j' : System.exit(0);
            break;
            default :
            break;
        }
    }

    /**
     * getIOMethod method
     *
     * @return      whether we are using RandomAccessFile or Sequential access for input and output.
     */
    public static boolean getIOMethod()
    {
        return withRandomAccessFile;
    }

    /**
     * listAllBooks method - lists all the publishers to the terminal.
     */
    public void listAllPublishers()
    {
        if(!getIOMethod()) {
            //sequentially
            PublisherFile.outputAllPublishersToTerminal();
        } else {
            //random access
            PublisherFileRAF.outputAllPublishersToTerminal();
        }

        pressToContinue();
    }

    /**
```

```
 * addPublisher method - allows the user to add a publisher through the terminal
 */
public void addPublisher()
{
    Publisher p = InputTerminal.publisherEntry();

    if(!Bookstore.getIOMethod()) {
        //sequentially
        PublisherFile.writeToPublisherFile(p);
    } else {
        //random access
        PublisherFileRAF.writeToPublisherFile(p);
    }

    pressToContinue();
}

/**
 * removePublisher method - allows the user to remove a publisher through the terminal
 */
public void removePublisher()
{
    System.out.println("");
    System.out.println("Search for a publisher to remove by:");
    System.out.println("a) ID");
    System.out.println("b) Name");
    System.out.println("");

    char selection;

    do {
        selection = IO.input("Which action do you want to perform? ").charAt(0);
    } while(selection != 'a' && selection != 'b');

    Publisher[] publishers = null;

    if(!getIOMethod()) {
        //sequentially
        PublisherFile.outputAllPublishersIDsNamesToTerminal();
        publishers = PublisherFile.readPublisherFile();
    } else {
        //random access
        PublisherFileRAF.outputAllPublishersIDsNamesToTerminal();
```

```
            publishers = PublisherFileRAF.readPublisherFile();
        }

        int toRemove = -1;

        if(selection == 'a') {
            int searchParameter = Integer.valueOf(IO.input("ID to search: ").trim()).intValue();
            toRemove = LinearSearch.linearSearch(searchParameter, publishers);
        } else {
            String searchParameter = IO.input("Name to search:");
            toRemove = LinearSearch.linearSearch(searchParameter, publishers);
        }

        if(toRemove == -1)
            System.out.println("There is no such publisher.");
        else {
            int id = publishers[toRemove].getId();
            String name = publishers[toRemove].getName();
            String address = publishers[toRemove].getAddress();
            int telephone = publishers[toRemove].getTelephone();
            String website = publishers[toRemove].getWebsite();

            System.out.println("");
            System.out.println("==================================");
            System.out.println("ID:" + name);
            System.out.println("NAME:" + name);
            System.out.println("ADDRESS:" + address);
            System.out.println("TELEPHONE:" + telephone);
            System.out.println("WEBSITE:" + website);
            System.out.println("==================================");
            System.out.println("");

            do {
                selection = IO.input("Are you sure you want to remove the above book entry, Yes (y) or No (n)?
").charAt(0);
            } while(selection != 'y' && selection != 'Y' && selection != 'n' && selection != 'N');

            if(selection == 'y' || selection == 'Y') {

                Publisher[] temp = new Publisher[publishers.length-1];

                int j = 0;
```

```
                for(int i=0; i<publishers.length; i++) {
                    if(toRemove != i) {
                        temp[j] = publishers[i];
                        j = j+1;
                    }
                }

                if(!getIOMethod()) {
                    //sequentially
                    PublisherFile.writePublishersToPublisherFile(temp);
                } else {
                    //random access
                    PublisherFileRAF.rewriteToPublisherFile(temp);
                }

                System.out.println("Publisher entry deleted.");
            }
        }

        pressToContinue();
    }

    /**
     * listAllReadingMaterial method - lists all the reading material of the bookstore in the terminal.
     */
    public void listAllReadingMaterial()
    {
        System.out.println("");
        System.out.println("List all reading materials of type:");
        System.out.println("a) Books");
        System.out.println("b) Magazines");
        System.out.println("");

        char selection;

        do {
            selection = IO.input("Which action do you want to perform? ").charAt(0);
        } while(selection != 'a' && selection != 'b');

        if(selection == 'a') {
            if(!getIOMethod()) {
                //sequentially
```

```
            ReadingMaterialFile.outputAllBooksToTerminal();
        } else {
            //random access
            ReadingMaterialFileRAF.outputAllBooksToTerminal();
        }

    } else {
        if(!getIOMethod()) {
            //sequentially
            ReadingMaterialFile.outputAllMagazinesToTerminal();
        } else {
            //random access
            ReadingMaterialFileRAF.outputAllMagazinesToTerminal();
        }
    }

    pressToContinue();
}

/**
 * listReadingMaterialByPublisher method - lists all the reading material, of a certain
 * publisher, of the bookstore in the terminal.
 */
public void listReadingMaterialByPublisher()
{
    System.out.println("");

    if(!getIOMethod()) {
        //sequentially
        PublisherFile.outputAllPublishersIDsNamesToTerminal();
    } else {
        //random access
        PublisherFileRAF.outputAllPublishersIDsNamesToTerminal();
    }

    System.out.println("");

    int id = Integer.valueOf(IO.input("List all reading materials of which publisher (ID)? ").trim()).intValue();
    while(id < 0 || id == 0) {
        System.out.println("Error: The id you entered was not a proper id number");
        id = Integer.valueOf(IO.input("List all reading materials of which publisher (ID)? ").trim()).intValue();
    }
```

```
        System.out.println("");
        System.out.println("Books");
        System.out.println("---------");

        if(!getIOMethod()) {
            //sequentially
            ReadingMaterialFile.outputAllBooksToTerminalOfPublisher(id);

        } else {
            //random access
            ReadingMaterialFileRAF.outputAllBooksToTerminalOfPublisher(id);
        }

        System.out.println("");
        System.out.println("Magazines");
        System.out.println("---------");

        if(!getIOMethod()) {
            //sequentially
            ReadingMaterialFile.outputAllMagazinesToTerminalOfPublisher(id);

        } else {
            //random access
            ReadingMaterialFileRAF.outputAllMagazinesToTerminalOfPublisher(id);
        }

        pressToContinue();
    }

    /**
     * addReadingMaterial method - allows the user to add a reading material through the terminal
     */
    public void addReadingMaterial()
    {
        System.out.println("");
        System.out.println("What type of reading material to add:");
        System.out.println("a) Book");
        System.out.println("b) Magazine");
        System.out.println("");
```

```
    char selection;

    do {
        selection = IO.input("Which action do you want to perform? ").charAt(0);
    } while(selection != 'a' && selection != 'b');

    if(selection == 'a') {
        Book b = InputTerminal.bookEntry();

        if(!Bookstore.getIOMethod()) {
            //sequentially
            ReadingMaterialFile.writeToBookFile(b);
        } else {
            //random access
            ReadingMaterialFileRAF.writeToBookFile(b);
        }
    } else {
        Magazine m = InputTerminal.magazineEntry();

        if(!Bookstore.getIOMethod()) {
            //sequentially
            ReadingMaterialFile.writeToMagazineFile(m);
        } else {
            //random access
            ReadingMaterialFileRAF.writeToMagazineFile(m);
        }
    }

    pressToContinue();
}

/**
 * removeReadingMaterial method - allows the user to remove a book through the terminal
 */
public void removeReadingMaterial()
{
    System.out.println("");
    System.out.println("What type of reading material to remove:");
    System.out.println("a) Book");
    System.out.println("b) Magazine");
```

```
System.out.println("");

char selection;

do {
    selection = IO.input("Which action do you want to perform? ").charAt(0);
} while(selection != 'a' && selection != 'b');

if(selection == 'a') {

    Book[] books = loadBooks();
    int toRemove = searchForReadingMaterial(books, true);

    if(toRemove == -1)
        System.out.println("There is no such book in the bookstore.");
    else {
        int id = books[toRemove].getId();
        String title = books[toRemove].getTitle();
        int pages = books[toRemove].getPages();
        int price = books[toRemove].getPrice();
        int chapters = books[toRemove].getChapters();
        Publisher publisher = books[toRemove].getPublisher();

        System.out.println("");
        System.out.println("==================================");
        System.out.println("ID:" + id);
        System.out.println("TITLE:" + title);
        System.out.println("PAGES:" + pages);
        System.out.println("PRICE ($):" + price);
        System.out.println("CHAPTERS:" + chapters);

        String[] authors = books[toRemove].getAuthors();

        for(int j=0; j<authors.length; j++) {
            if(!(authors[j].trim().equals("")))
                System.out.println("AUTHOR(S):" + authors[j]);
        }

        System.out.println("PUBLISHER:" + publisher.getName());
        System.out.println("==================================");
        System.out.println("");
```

```
            do {
                selection = IO.input("Are you sure you want to remove the above book entry, Yes (y) or No (n)? ").charAt(0);
            } while(selection != 'y' && selection != 'Y' && selection != 'n' && selection != 'N');

            if(selection == 'y' || selection == 'Y') {

                Book[] temp = new Book[books.length-1];

                int j = 0;

                for(int i=0; i<books.length; i++) {
                    if(toRemove != i) {
                        temp[j] = books[i];
                        j = j+1;
                    }
                }

                if(!getIOMethod()) {
                    //sequentially
                    ReadingMaterialFile.writeBooksToBookFile(temp);
                } else {
                    //random access
                    ReadingMaterialFileRAF.rewriteToBookFile(temp);
                }

                System.out.println("Book entry deleted.");
            }
        }
    } else {

        Magazine[] magazines = loadMagazines();
        int toRemove = searchForReadingMaterial(magazines, true);

        if(toRemove == -1)
            System.out.println("There is no such magazine in the bookstore.");
        else {
            int id = magazines[toRemove].getId();
            String title = magazines[toRemove].getTitle();
            int pages = magazines[toRemove].getPages();
            int price = magazines[toRemove].getPrice();

            Publisher publisher = magazines[toRemove].getPublisher();
```

```
System.out.println("");
System.out.println("==================================");
System.out.println("ID:" + id);
System.out.println("TITLE:" + title);
System.out.println("PAGES:" + pages);
System.out.println("PRICE ($):" + price);

String[] resources = magazines[toRemove].getAdditionalResources();

for(int j=0; j<resources.length; j++) {
    if(!(resources[j].trim().equals("")))
        System.out.println("RESOURCE(S):" + resources[j]);
}

System.out.println("PUBLISHER:" + publisher.getName());
System.out.println("==================================");
System.out.println("");

do {
    selection = IO.input("Are you sure you want to remove the above magazine entry, Yes (y) or No (n)? ").charAt(0);
} while(selection != 'y' && selection != 'Y' && selection != 'n' && selection != 'N');

if(selection == 'y' || selection == 'Y') {

    Magazine[] temp = new Magazine[magazines.length-1];

    int j = 0;

    for(int i=0; i<magazines.length; i++) {
        if(toRemove != i) {
            temp[j] = magazines[i];
            j = j+1;
        }
    }

    if(!getIOMethod()) {
        //sequentially
        ReadingMaterialFile.writeMagazinesToMagazineFile(temp);
    } else {
        //random access
        ReadingMaterialFileRAF.rewriteToMagazineFile(temp);
    }
```

```
                System.out.println("Magazine entry deleted.");
            }
        }
    }

    pressToContinue();
}

/**
 * sortReadingMaterialByPrice method - Displays the reading material sorted by price
 *
 */
public void sortReadingMaterialByPrice()
{
    System.out.println("");
    System.out.println("What type of reading material to sort by price:");
    System.out.println("a) Book");
    System.out.println("b) Magazine");
    System.out.println("");

    char selection;

    do {
        selection = IO.input("Which action do you want to perform? ").charAt(0);
    } while(selection != 'a' && selection != 'b');

    System.out.println("");
    System.out.println("What type of sorting:");
    System.out.println("a) Ascending");
    System.out.println("b) Descending");
    System.out.println("");

    char sortType;

    do {
        sortType = IO.input("Which action do you want to perform? ").charAt(0);
    } while(sortType != 'a' && sortType != 'b');

    if(selection == 'a') {
        Book[] books = loadBooks();
```

```
if(books == null || books.length == 0) {
    System.out.println("There are no books in the bookstore to sort.");
} else {
    if(sortType == 'a') {
        books = SelectionSort.selectionSort(books, true);
        System.out.println("");
        System.out.println("Sorted books by price (ascending):");
        System.out.println("");
    } else {
        books = SelectionSort.selectionSort(books, false);
        System.out.println("");
        System.out.println("Sorted books by price (descending):");
        System.out.println("");
    }

    for(int i=books.length-1; i>=0; i--) {
        int id = books[i].getId();
        String title = books[i].getTitle();
        int pages = books[i].getPages();
        int price = books[i].getPrice();
        int chapters = books[i].getChapters();
        Publisher publisher = books[i].getPublisher();

        System.out.println("");
        System.out.println("==================================");
        System.out.println("ID:" + id);
        System.out.println("TITLE:" + title);
        System.out.println("PAGES:" + pages);
        System.out.println("PRICE ($):" + price);
        System.out.println("CHAPTERS:" + chapters);

        String[] authors = books[i].getAuthors();

        for(int j=0; j<authors.length; j++) {
            if(!(authors[j].trim().equals("")))
                System.out.println("AUTHOR(S):" + authors[j]);
        }

        System.out.println("PUBLISHER:" + publisher.getName());
        System.out.println("==================================");
```

```
                System.out.println("");
            }
        }
    } else {
        Magazine[] magazines = loadMagazines();

        if(magazines == null || magazines.length == 0) {
            System.out.println("There are no magazines in the bookstore to sort.");
        } else {
            if(sortType == 'a') {
                magazines = SelectionSort.selectionSort(magazines, true);
                System.out.println("");
                System.out.println("Sorted magazines by price (ascending):");
                System.out.println("");
            } else {
                magazines = SelectionSort.selectionSort(magazines, false);
                System.out.println("");
                System.out.println("Sorted magazines by price (descending):");
                System.out.println("");
            }

            for(int i=magazines.length-1; i>=0; i--) {
                int id = magazines[i].getId();
                String title = magazines[i].getTitle();
                int pages = magazines[i].getPages();
                int price = magazines[i].getPrice();

                Publisher publisher = magazines[i].getPublisher();

                System.out.println("");
                System.out.println("==================================");
                System.out.println("ID:" + id);
                System.out.println("TITLE:" + title);
                System.out.println("PAGES:" + pages);
                System.out.println("PRICE ($):" + price);

                String[] resources = magazines[i].getAdditionalResources();

                for(int j=0; j<resources.length; j++) {
                    if(!(resources[j].trim().equals("")))
                        System.out.println("RESOURCE(S):" + resources[j]);
```

```
                }

                System.out.println("PUBLISHER:" + publisher.getName());
                System.out.println("==================================");
                System.out.println("");
            }
        }
    }

    pressToContinue();
}

/**
 * findReadingMaterial method - Finds a reading material
 */
public void findReadingMaterial()
{
    System.out.println("");
    System.out.println("What type of reading material to find:");
    System.out.println("a) Book");
    System.out.println("b) Magazine");
    System.out.println("");

    char selection;

    do {
        selection = IO.input("Which action do you want to perform? ").charAt(0);
    } while(selection != 'a' && selection != 'b');

    if(selection == 'a') {

        Book[] books = loadBooks();
        int toRemove = searchForReadingMaterial(books, true);

        if(toRemove == -1)
            System.out.println("There is no such book in the bookstore.");
        else {
            int id = books[toRemove].getId();
            String title = books[toRemove].getTitle();
            int pages = books[toRemove].getPages();
            int price = books[toRemove].getPrice();
            int chapters = books[toRemove].getChapters();
            Publisher publisher = books[toRemove].getPublisher();
```

```
            System.out.println("");
            System.out.println("==================================");
            System.out.println("ID:" + id);
            System.out.println("TITLE:" + title);
            System.out.println("PAGES:" + pages);
            System.out.println("PRICE ($):" + price);
            System.out.println("CHAPTERS:" + chapters);

            String[] authors = books[toRemove].getAuthors();

            for(int j=0; j<authors.length; j++) {
                if(!(authors[j].trim().equals("")))
                    System.out.println("AUTHOR(S):" + authors[j]);
            }

            System.out.println("PUBLISHER:" + publisher.getName());
            System.out.println("==================================");
            System.out.println("");
        }
    } else {
        Magazine[] magazines = loadMagazines();
        int toRemove = searchForReadingMaterial(magazines, true);

        if(toRemove == -1)
            System.out.println("There is no such magazine in the bookstore.");
        else {
            int id = magazines[toRemove].getId();
            String title = magazines[toRemove].getTitle();
            int pages = magazines[toRemove].getPages();
            int price = magazines[toRemove].getPrice();

            Publisher publisher = magazines[toRemove].getPublisher();

            System.out.println("");
            System.out.println("==================================");
            System.out.println("ID:" + id);
            System.out.println("TITLE:" + title);
            System.out.println("PAGES:" + pages);
            System.out.println("PRICE ($):" + price);

            String[] resources = magazines[toRemove].getAdditionalResources();
```

```
            for(int j=0; j<resources.length; j++) {
                if(!(resources[j].trim().equals("")))
                    System.out.println("RESOURCE(S):" + resources[j]);
            }

            System.out.println("PUBLISHER:" + publisher.getName());
            System.out.println("==================================");
            System.out.println("");
        }
    }

    pressToContinue();
}

/**
 * searchForReadingMaterial method - allows the user to search for a book
 * through the terminal.
 *
 * @param books  an array of available books
 * @param displayAllBooks  a boolean signifying whether or not to display all books available
 *
 * @return an integer signifying the index of the book searched for in the books array. -1 is no book found.
 */
private int searchForReadingMaterial(Book[] books, boolean displayAllBooks) {
    System.out.println("");
    System.out.println("Search for a book to remove by:");
    System.out.println("a) ID");
    System.out.println("b) Title");
    System.out.println("");

    char selection;

    do {
        selection = IO.input("Which action do you want to perform? ").charAt(0);
    } while(selection != 'a' && selection != 'b');

    int toRemove = -1;

    System.out.println();

    if(displayAllBooks) {
```

```
            if(!getIOMethod()) {
                //sequentially
                ReadingMaterialFile.outputAllBooksIDsTitlesToTerminal();
            } else {
                //random access
                ReadingMaterialFileRAF.outputAllBooksIDsTitlesToTerminal();
            }
        }

        if(selection == 'a') {

            int searchParameter = Integer.valueOf(IO.input("ID to search: ").trim()).intValue();
            toRemove = LinearSearch.linearSearch(searchParameter, books);
        } else {

            String searchParameter = IO.input("Title to search:");
            toRemove = LinearSearch.linearSearch(searchParameter, books);
        }

        return toRemove;
    }

    /**
     * searchForReadingMaterial method - allows the user to search for a magazine
     * through the terminal.
     *
     * @param magazines  an array of available magazines
     * @param displayAllMagazines  a boolean signifying whether or not to display all magazines available
     *
     * @return an integer signifying the index of the magazine searched for in the magazines array.
     * -1 is no book found.
     */
    private int searchForReadingMaterial(Magazine[] magazines, boolean displayAllMagazines) {
        System.out.println("");
        System.out.println("Search for a magazine to remove by:");
        System.out.println("a) ID");
        System.out.println("b) Title");
        System.out.println("");

        char selection;
```

```
        do {
            selection = IO.input("Which action do you want to perform? ").charAt(0);
        } while(selection != 'a' && selection != 'b');

        int toRemove = -1;

        System.out.println();

        if(displayAllMagazines) {

            if(!getIOMethod()) {
                //sequentially
                ReadingMaterialFile.outputAllMagazinesIDsTitlesToTerminal();
            } else {
                //random access
                ReadingMaterialFileRAF.outputAllMagazinesIDsTitlesToTerminal();
            }
        }

        if(selection == 'a') {

            int searchParameter = Integer.valueOf(IO.input("ID to search: ").trim()).intValue();
            toRemove = LinearSearch.linearSearch(searchParameter, magazines);
        } else {

            String searchParameter = IO.input("Title to search:");
            toRemove = LinearSearch.linearSearch(searchParameter, magazines);
        }

        return toRemove;
    }

    /**
     * loadBooks method - loads books from the file.
     *
     * @return an array of books.
     */
    private Book[] loadBooks() {
        Book[] books = null;

        if(!getIOMethod()) {
            //sequentially
```

```
            books = ReadingMaterialFile.readBookFile();
        } else {
            //random access
            books = ReadingMaterialFileRAF.readBookFile();
        }

        return books;
    }

    /**
     * loadMagazines method - loads magazines from the file.
     *
     * @return an array of magazines.
     */
    private Magazine[] loadMagazines() {
        Magazine[] magazines = null;

        if(!getIOMethod()) {
            //sequentially
            magazines = ReadingMaterialFile.readMagazineFile();
        } else {
            //random access
            magazines = ReadingMaterialFileRAF.readMagazineFile();
        }

        return magazines;
    }

    /**
     * pressToContinue method - displays message and continues to menu.
     */
    private void pressToContinue() {
        System.out.println("");
        IO.input("Press ANY BUTTON to continue.");
        System.out.println("");
        showMenu();
    }
}
```

```
//ReadingMaterialFileRAF

import java.io.RandomAccessFile;
import java.io.IOException;

public class ReadingMaterialFileRAF
{
    /*
     * STRING_BYTES represents the length of any String input, in bytes.
     * 25 Bytes mean that the Strings used in this program, to store
     * various details (ex. titles, authors, etc.), may have a maximum of
     * 25 characters each. This way, every String that will
     * be used in this program will take up 25 bytes in the output files.
     * This is decided by the programmer. It could have been any number.
     */
    static final int STRING_BYTES = 25;

    //INT_BYTES represents the Bytes needed to store an int in the output files.
    static final int INT_BYTES = 4;

    /*
     * BOOK_RECORD_BYTES represents a book record stored in a file.
     * Since a book record is made up of:
     *
     * 5 ints (id, pages, price, chapters, publisher id)
     * 5 Strings (title, author 1, author 2, author 3, author 4)
     *
     * each book record stored in file would take up:
     *
     * 4*5 + 25*5 = 16 + 125 = 145 bytes
     *
     * It is assumed that a maximum of four authors can be present for any book.
     */
    static final int BOOK_RECORD_BYTES = INT_BYTES*5 + STRING_BYTES*5;
    /*
     * MAGAZINE_RECORD_BYTES is similar to BOOK_RECORD_BYTES but is made
     * up of:
     *
     * 4 ints (id, pages, price, publisher id)
```

```
 * 5 Strings (title, resource 1, resource 2, resource 3, resource 4)
 *
 * each magazine record stored in file would take up:
 *
 * 4*4 + 25*5 = 141 bytes
 *
 * It is assumed that a maximum of four additional resources can be present
 * for any magazine.
 */
static final int MAGAZINE_RECORD_BYTES = INT_BYTES*4 + STRING_BYTES*5;

static final String BOOK_FILENAME = "bookFileRAF";
static final String MAGAZINE_FILENAME = "magazineFileRAF";

//Book related methods.

/**
 * readBookFile - reads all book entries from the bookstore
 *
 * @return     array with all the contents of the bookFile
 */
public static Book[] readBookFile()
{
    RandomAccessFile theBookFile;
    int numberOfEntries = 0;

    try{
        theBookFile = new RandomAccessFile(BOOK_FILENAME, "rw");
        int bookFileSize = (int) theBookFile.length();

        numberOfEntries = bookFileSize / BOOK_RECORD_BYTES;

        Book[] bookEntries = new Book[numberOfEntries];
        try{
            //loop through the datafile
            for(int i=0; i<numberOfEntries; i++) {
                int entryToRead = i*BOOK_RECORD_BYTES;

                bookEntries[i] = readBookFileRecord(entryToRead);
            }
```

```
            } catch(Exception e) {
                //bookFile does not exist or can not be created.
                String err = e.toString();
                System.out.println("");
                System.out.println("The book file does not exist or can not be created.");
                System.out.println("");
            }

            return bookEntries;

        } catch (IOException io) {
            System.out.println("Error trying to open the bookstore file.");
        }

        return null;
    }

    /**
     * readBookFileRecord - reads a book entry from the bookstore
     *
     * @param entryToRead   number of entry to read from the bookFile
     */
    public static Book readBookFileRecord(int entryToRead) throws IOException
    {
        RandomAccessFile theBookFile;
        int numberOfEntries = 0;

        try{
            theBookFile = new RandomAccessFile(BOOK_FILENAME, "rw");
            int bookFileSize = (int) theBookFile.length();
            numberOfEntries = bookFileSize / BOOK_RECORD_BYTES;

            theBookFile.seek(entryToRead);
            int id = theBookFile.readInt();

            byte[] titleBytes = new byte[STRING_BYTES];
            theBookFile.readFully(titleBytes);
            String title = new String(titleBytes);

            int pages = theBookFile.readInt();
```

```
            int price = theBookFile.readInt();
            int chapters = theBookFile.readInt();
            int publisherID = theBookFile.readInt();

            byte[] author1Bytes = new byte[STRING_BYTES];
            theBookFile.readFully(author1Bytes);
            String author1 = new String(author1Bytes);

            byte[] author2Bytes = new byte[STRING_BYTES];
            theBookFile.readFully(author2Bytes);
            String author2 = new String(author2Bytes);

            byte[] author3Bytes = new byte[STRING_BYTES];
            theBookFile.readFully(author3Bytes);
            String author3 = new String(author3Bytes);

            byte[] author4Bytes = new byte[STRING_BYTES];
            theBookFile.readFully(author4Bytes);
            String author4 = new String(author4Bytes);

            String[] authors = { author1, author2, author3, author4 };

            Publisher publisher = PublisherFile.readPublisherWithID(publisherID);

            return new Book(id, title, pages, price, chapters, authors, publisher);
        } catch (IOException io) {
            System.out.println("Error trying to open the bookstore file.");
        }

        return null;
    }

    /**
     * writeToBookFile - writes a book entry to the bookstore
     *
     * @param entry          entry to write to the bookFile
     * @return               whether the write was successful
     */
    public static Boolean writeToBookFile(Book entry)
    {
        RandomAccessFile theBookFile;
```

```
    int numberOfEntries = 0;

    try{
        theBookFile = new RandomAccessFile(BOOK_FILENAME, "rw");
        int bookFileSize = (int) theBookFile.length();
        numberOfEntries = bookFileSize / BOOK_RECORD_BYTES;

        try {
            int entryToWrite = numberOfEntries*BOOK_RECORD_BYTES;
            writeBookFileRecord(entryToWrite, entry);
            numberOfEntries = numberOfEntries+1;
            return true;

        } catch (IOException io) {
            System.out.println("Error while writing to the bookstore.");

            return false;
        }
    } catch (IOException io) {
        System.out.println("Error trying to open the bookstore file.");
    }

    return null;
}

 /**
  * rewriteToBookFile - re-writes book entries to the bookstore
  *
  * @param entries          entries to write to the bookFile
  * @return                 whether the write was successful
  */
public static Boolean rewriteToBookFile(Book[] entries)
{
    RandomAccessFile theBookFile;
    int numberOfEntries = 0;

    try{
        theBookFile = new RandomAccessFile(BOOK_FILENAME, "rw");
        int bookFileSize = (int) theBookFile.length();
        numberOfEntries = bookFileSize / BOOK_RECORD_BYTES;
```

```
            try {
                theBookFile.setLength(0);

                numberOfEntries = entries.length;

                for(int i=0; i<entries.length; i++) {
                    int entryToWrite = i*BOOK_RECORD_BYTES;
                    writeBookFileRecord(entryToWrite, entries[i]);
                }
                return true;

            } catch (IOException io) {
                System.out.println("Error while writing to the bookstore.");

                return false;
            }
        } catch (IOException io) {
            System.out.println("Error trying to open the bookstore file.");
        }

        return null;
    }

    /**
     * writeBookFileRecord - writes a book entry to the bookstore
     *
     * @param entryToWrite   number of entry to write to the bookFile
     * @param entry          entry to write to the bookFile
     */
    public static void writeBookFileRecord(int entryToWrite, Book entry) throws IOException
    {
        RandomAccessFile theBookFile;
        int numberOfEntries = 0;

        try{
            theBookFile = new RandomAccessFile(BOOK_FILENAME, "rw");
            int bookFileSize = (int) theBookFile.length();
            numberOfEntries = bookFileSize / BOOK_RECORD_BYTES;

            int id = entry.getId();
```

```
String title = entry.getTitle();
StringBuffer sbt = new StringBuffer(title);

while(sbt.length() < STRING_BYTES) {
    sbt.append(" ");
}

title = sbt.toString();

int pages = entry.getPages();
int price = entry.getPrice();
int chapters = entry.getChapters();
int publisherID = entry.getPublisher().getId();

theBookFile.seek(entryToWrite);
theBookFile.writeInt(id);
theBookFile.write(title.getBytes());
theBookFile.writeInt(pages);
theBookFile.writeInt(price);
theBookFile.writeInt(chapters);
theBookFile.writeInt(publisherID);

String[] authors = entry.getAuthors();

for(int i=0; i<authors.length; i++) {
    StringBuffer sba = new StringBuffer(authors[i]);

    while(sba.length() < STRING_BYTES) {
        sba.append(" ");
    }

    theBookFile.writeBytes(sba.toString());
}

for(int i=authors.length; i<4; i++) {
    StringBuffer sba = new StringBuffer("");

    while(sba.length() < STRING_BYTES) {
        sba.append(" ");
```

```
            }

            theBookFile.writeBytes(sba.toString());
        }
    } catch (IOException io) {
        System.out.println("Error trying to open the bookstore file.");
    }
}

/**
 * outputAllBooksToTerminal - outputs all the books in the bookstore to the terminal without sorting
 *
 */
public static void outputAllBooksToTerminal()
{
    Book[] books = null;
    books = readBookFile();

    if(books != null) {

        for(int i=0; i < books.length; i++) {

            Book temp = books[i];

            int id = temp.getId();
            String title = temp.getTitle();
            int pages = temp.getPages();
            int price = temp.getPrice();
            int chapters = temp.getChapters();
            Publisher publisher = temp.getPublisher();

            System.out.println("");
            System.out.println("================================");
            System.out.println("ID:" + id);
            System.out.println("TITLE:" + title);
            System.out.println("PAGES:" + pages);
            System.out.println("PRICE ($):" + price);
            System.out.println("CHAPTERS:" + chapters);

            String[] authors = temp.getAuthors();
```

```
            for(int j=0; j<authors.length; j++) {
                if(!(authors[j].trim().equals("")))
                    System.out.println("AUTHOR(S):" + authors[j]);
                }

                System.out.println("PUBLISHER: " + publisher.getName());
                System.out.println("==================================");
                System.out.println("");
            }

        } else {
            System.out.println("");
            System.out.println("There are no books in the bookstore.");
            System.out.println("");
        }
    }

    /**
     * outputAllBooksIDsTitlesToTerminal - outputs all the books' IDs and titles
     * in the bookstore to the terminal without sorting
     *
     */
    public static void outputAllBooksIDsTitlesToTerminal()
    {
        Book[] books = null;
        books = readBookFile();

        if(books != null) {
            if(books.length == 0) {
                System.out.println("");
                System.out.println("The bookstore is empty.");
                System.out.println("");
            } else {

                System.out.println("");
                System.out.println("Available books.");
                System.out.println("==================================");

                for(int i=0; i < books.length; i++) {
```

```
                Book temp = books[i];

                int id = temp.getId();
                String title = temp.getTitle();
                int pages = temp.getPages();
                int price = temp.getPrice();
                int chapters = temp.getChapters();
                Publisher publisher = temp.getPublisher();

                System.out.println("ID:" + id + " TITLE:" + title);

            }

            System.out.println("==================================");
            System.out.println("");
        }

    } else {
        System.out.println("");
        System.out.println("There are no books in the bookstore.");
        System.out.println("");
    }
}

/**
 * outputAllBooksToTerminalOfPublisher - outputs all the books in the bookstore,
 * of a certain publisher, to the terminal without sorting
 */
public static void outputAllBooksToTerminalOfPublisher(int target)
{
    Book[] books = null;
    books = readBookFile();

    if(books != null) {
        if(books.length == 0) {
            System.out.println("");
            System.out.println("The bookstore is empty.");
            System.out.println("");
        } else {
```

```
int booksFound = 0;

for(int i=0; i < books.length; i++) {

    Book temp = books[i];

    int id = temp.getId();
    String title = temp.getTitle();
    int pages = temp.getPages();
    int price = temp.getPrice();
    int chapters = temp.getChapters();
    Publisher publisher = temp.getPublisher();

    if(target == publisher.getId()) {

        booksFound = booksFound + 1;

        System.out.println("");
        System.out.println("==================================");
        System.out.println("ID:" + id);
        System.out.println("TITLE:" + title);
        System.out.println("PAGES:" + pages);
        System.out.println("PRICE ($):" + price);

        System.out.println("CHAPTERS:" + chapters);

        String[] authors = temp.getAuthors();
        for(int j=0; j<authors.length; j++) {

            if(!(authors[j].trim().equals("")))
                System.out.println("AUTHOR(S):" + authors[j]);
        }

        System.out.println("PUBLISHER:" + publisher.getName());
        System.out.println("==================================");
        System.out.println("");
    }

    if(booksFound == 0) {
        System.out.println("No books found from this publisher.");
```

```
                }
            }
        }

    } else {
        System.out.println("");
        System.out.println("There are no books in the bookstore.");
        System.out.println("");
    }
}

//Magazine related methods. - To be completed as exercise.

 /**
  * readMagazineFile - reads all magazine entries from the bookstore
  *
  */
public static Magazine[] readMagazineFile()
{
    RandomAccessFile theMagazineFile;
    int numberOfEntries = 0;

    try{
        theMagazineFile = new RandomAccessFile(MAGAZINE_FILENAME, "rw");
        int magazineFileSize = (int) theMagazineFile.length();

        numberOfEntries = magazineFileSize / MAGAZINE_RECORD_BYTES;

        Magazine[] magazineEntries = new Magazine[numberOfEntries];
        try{
            //loop through the datafile
            for(int i=0; i<numberOfEntries; i++) {
                int entryToRead = i*MAGAZINE_RECORD_BYTES;

                magazineEntries[i] = readMagazineFileRecord(entryToRead);
            }
        } catch(Exception e) {
            //magazineFile does not exist or can not be created.
            String err = e.toString();
            System.out.println("");
```

```
            System.out.println("The magazine file does not exist or can not be created.");
            System.out.println("");
        }

        return magazineEntries;

    } catch (IOException io) {
        System.out.println("Error trying to open the bookstore file.");
    }

    return null;
}

/**
 * readMagazineFileRecord - reads a magazine entry from the bookstore
 *
 * @param entryToRead   number of entry to read from the magazineFile
 */
public static Magazine readMagazineFileRecord(int entryToRead) throws IOException
{
    RandomAccessFile theMagazineFile;
    int numberOfEntries = 0;

    try{
        theMagazineFile = new RandomAccessFile(MAGAZINE_FILENAME, "rw");
        int magazineFileSize = (int) theMagazineFile.length();
        numberOfEntries = magazineFileSize / MAGAZINE_RECORD_BYTES;

        theMagazineFile.seek(entryToRead);
        int id = theMagazineFile.readInt();

        byte[] titleBytes = new byte[STRING_BYTES];
        theMagazineFile.readFully(titleBytes);
        String title = new String(titleBytes);

        int pages = theMagazineFile.readInt();
        int price = theMagazineFile.readInt();
        int publisherID = theMagazineFile.readInt();

        byte[] resource1Bytes = new byte[STRING_BYTES];
```

```
            theMagazineFile.readFully(resource1Bytes);
            String resource1 = new String(resource1Bytes);

            byte[] resource2Bytes = new byte[STRING_BYTES];
            theMagazineFile.readFully(resource2Bytes);
            String resource2 = new String(resource2Bytes);

            byte[] resource3Bytes = new byte[STRING_BYTES];
            theMagazineFile.readFully(resource3Bytes);
            String resource3 = new String(resource3Bytes);

            byte[] resource4Bytes = new byte[STRING_BYTES];
            theMagazineFile.readFully(resource4Bytes);
            String resource4 = new String(resource4Bytes);

            String[] resources = { resource1, resource2, resource3, resource4 };

            Publisher publisher = PublisherFile.readPublisherWithID(publisherID);

            return new Magazine(id, title, pages, price, resources, publisher);
        } catch (IOException io) {
            System.out.println("Error trying to open the bookstore file.");
        }

        return null;
    }

    /**
     * writeToMagazineFile - writes a book magazine to the bookstore
     *
     * @param entry          entry to write to the magazineFile
     * @return               whether the write was successful
     */
    public static Boolean writeToMagazineFile(Magazine entry)
    {
        RandomAccessFile theMagazineFile;
        int numberOfEntries = 0;

        try{
            theMagazineFile = new RandomAccessFile(MAGAZINE_FILENAME, "rw");
            int magazineFileSize = (int) theMagazineFile.length();
```

```
            numberOfEntries = magazineFileSize / MAGAZINE_RECORD_BYTES;

            try {
                int entryToWrite = numberOfEntries*MAGAZINE_RECORD_BYTES;
                writeMagazineFileRecord(entryToWrite, entry);
                numberOfEntries = numberOfEntries+1;
                return true;

            } catch (IOException io) {
                System.out.println("Error while writing to the bookstore.");

                return false;
            }
        } catch (IOException io) {
            System.out.println("Error trying to open the bookstore file.");
        }

        return null;
    }

    /**
     * rewriteToMagazineFile - re-writes magazine entries to the bookstore
     *
     * @param entries           entries to write to the magazineFile
     * @return                  whether the write was successful
     */
    public static Boolean rewriteToMagazineFile(Magazine[] entries)
    {
        RandomAccessFile theMagazineFile;
        int numberOfEntries = 0;

        try{
            theMagazineFile = new RandomAccessFile(MAGAZINE_FILENAME, "rw");
            int magazineFileSize = (int) theMagazineFile.length();
            numberOfEntries = magazineFileSize / MAGAZINE_RECORD_BYTES;

            try {
                theMagazineFile.setLength(0);

                numberOfEntries = entries.length;
```

```
            for(int i=0; i<entries.length; i++) {
                int entryToWrite = i*MAGAZINE_RECORD_BYTES;
                writeMagazineFileRecord(entryToWrite, entries[i]);
            }
            return true;

        } catch (IOException io) {
            System.out.println("Error while writing to the bookstore.");

            return false;
        }
    } catch (IOException io) {
        System.out.println("Error trying to open the bookstore file.");
    }

    return null;
}

/**
 * writeMagazineFileRecord - writes a magazine entry to the bookstore
 *
 * @param entryToWrite   number of entry to write to the magazineFile
 * @param entry          entry to write to the magazineFile
 */
public static void writeMagazineFileRecord(int entryToWrite, Magazine entry) throws IOException
{
    RandomAccessFile theMagazineFile;
    int numberOfEntries = 0;

    try{
        theMagazineFile = new RandomAccessFile(MAGAZINE_FILENAME, "rw");
        int magazineFileSize = (int) theMagazineFile.length();
        numberOfEntries = magazineFileSize / MAGAZINE_RECORD_BYTES;

        int id = entry.getId();

        String title = entry.getTitle();
        StringBuffer sbt = new StringBuffer(title);

        while(sbt.length() < STRING_BYTES) {
            sbt.append(" ");
```

```
            }

            title = sbt.toString();

            int pages = entry.getPages();
            int price = entry.getPrice();
            int publisherID = entry.getPublisher().getId();

            theMagazineFile.seek(entryToWrite);
            theMagazineFile.writeInt(id);
            theMagazineFile.write(title.getBytes());
            theMagazineFile.writeInt(pages);
            theMagazineFile.writeInt(price);
            theMagazineFile.writeInt(publisherID);

            String[] resources = entry.getAdditionalResources();

            for(int i=0; i<resources.length; i++) {
                StringBuffer sba = new StringBuffer(resources[i]);

                while(sba.length() < STRING_BYTES) {
                    sba.append(" ");
                }

                theMagazineFile.writeBytes(sba.toString());
            }

            for(int i=resources.length; i<4; i++) {
                StringBuffer sba = new StringBuffer("");

                while(sba.length() < STRING_BYTES) {
                    sba.append(" ");
                }

                theMagazineFile.writeBytes(sba.toString());
            }
        } catch (IOException io) {
            System.out.println("Error trying to open the bookstore file.");
        }
    }

    /**
```

```
 * outputAllMagazinesToTerminal - outputs all the magazines in the bookstore
 * to the terminal without sorting
 */
public static void outputAllMagazinesToTerminal()
{
    Magazine[] magazines = null;
    magazines = readMagazineFile();

    if(magazines != null) {

        for(int i=0; i < magazines.length; i++) {

            Magazine temp = magazines[i];

            int id = temp.getId();
            String title = temp.getTitle();
            int pages = temp.getPages();
            int price = temp.getPrice();
            Publisher publisher = temp.getPublisher();

            System.out.println("");
            System.out.println("==================================");
            System.out.println("ID:" + id);
            System.out.println("TITLE:" + title);
            System.out.println("PAGES:" + pages);
            System.out.println("PRICE ($):" + price);

            String[] resources = temp.getAdditionalResources();

            for(int j=0; j<resources.length; j++) {
                if(!(resources[j].trim().equals("")))
                    System.out.println("RESOURCE(S):" + resources[j]);
            }

            System.out.println("PUBLISHER: " + publisher.getName());
            System.out.println("==================================");
            System.out.println("");
        }

    } else {
        System.out.println("");
```

```
            System.out.println("There are no books in the bookstore.");
            System.out.println("");
        }
    }

    /**
     * outputAllMagazinesIDsTitlesToTerminal - outputs all the magazines' IDs and titles
     * in the bookstore to the terminal without sorting
     *
     */
    public static void outputAllMagazinesIDsTitlesToTerminal()
    {
        Magazine[] magazines = null;
        magazines = readMagazineFile();

        if(magazines != null) {
            if(magazines.length == 0) {
                System.out.println("");
                System.out.println("The bookstore is empty.");
                System.out.println("");
            } else {

                System.out.println("");
                System.out.println("Available magazines.");
                System.out.println("==================================");

                for(int i=0; i < magazines.length; i++) {

                    Magazine temp = magazines[i];

                    int id = temp.getId();
                    String title = temp.getTitle();
                    int pages = temp.getPages();
                    int price = temp.getPrice();
                    Publisher publisher = temp.getPublisher();

                    System.out.println("ID:" + id + " TITLE:" + title);

                }

                System.out.println("==================================");
                System.out.println("");
            }
```

```
        } else {
            System.out.println("");
            System.out.println("There are no magazines in the bookstore.");
            System.out.println("");
        }
    }

    /**
     * outputAllMagazinesToTerminalOfPublisher - outputs all the magazines in the bookstore,
     * of a certain publisher, to the terminal without sorting
     */
    public static void outputAllMagazinesToTerminalOfPublisher(int target)
    {
        Magazine[] magazines = null;
        magazines = readMagazineFile();

        if(magazines != null) {
            if(magazines.length == 0) {
                System.out.println("");
                System.out.println("The bookstore is empty.");
                System.out.println("");
            } else {

                int magazinesFound = 0;

                for(int i=0; i < magazines.length; i++) {

                    Magazine temp = magazines[i];

                    int id = temp.getId();
                    String title = temp.getTitle();
                    int pages = temp.getPages();
                    int price = temp.getPrice();
                    Publisher publisher = temp.getPublisher();

                    if(target == publisher.getId()) {

                        magazinesFound = magazinesFound + 1;

                        System.out.println("");
                        System.out.println("================================");
                        System.out.println("ID:" + id);
```

```
                        System.out.println("TITLE:" + title);
                        System.out.println("PAGES:" + pages);
                        System.out.println("PRICE ($):" + price);

                        String[] resources = temp.getAdditionalResources();
                        for(int j=0; j<resources.length; j++) {

                            if(!(resources[j].trim().equals("")))
                              System.out.println("RESOURCE(S):" + resources[j]);
                        }

                        System.out.println("PUBLISHER:" + publisher.getName());
                        System.out.println("================================");
                        System.out.println("");
                    }

                    if(magazinesFound == 0) {
                        System.out.println("No magazines found from this publisher.");
                    }
                }
            }

        } else {
            System.out.println("");
            System.out.println("There are no books in the bookstore.");
            System.out.println("");
        }
    }
}

//PublisherFileRAF

import java.io.IOException;
import java.io.RandomAccessFile;

public class PublisherFileRAF
{
    /*
     * STRING_BYTES represents the length of any String input, in bytes.
     * 25 Bytes mean that the Strings used in this program, to store
     * various details (ex. name, address, etc.), may have a maximum of
     * 25 characters each. This way, every String that will
```

```
 * be used in this program will take up 25 bytes in the output file.
  * This is decided by the programmer. It could have been any number.
 */
static final int STRING_BYTES = 25;

//INT_BYTES represents the Bytes needed to store an int in the output file.
static final int INT_BYTES = 4;

 /*
 * RECORD_BYTES represents a publisher record stored in a file.
 * Since a publisher record is made up of:
 *
 * 2 ints (id, telephone)
 * 3 Strings (name, address, website)
 *
 * each book record stored in file would take up:
 *
 * 4*2 + 25*3 = 8 + 75 = 83 bytes
 */
static final int RECORD_BYTES = INT_BYTES*2 + STRING_BYTES*3;

static final String PUBLISHER_FILENAME = "publisherFileRAF";

 /**
 * readPublisherFile - reads all the publisher entries
 *
 * @return     array with all the contents of the publisherFile
 */
public static Publisher[] readPublisherFile()
{
    RandomAccessFile thePublisherFile;
    int numberOfEntries = 0;

    try{
        thePublisherFile = new RandomAccessFile(PUBLISHER_FILENAME, "rw");
        int publisherFileSize = (int) thePublisherFile.length();

        numberOfEntries = publisherFileSize / RECORD_BYTES;

        Publisher[] publisherEntries = new Publisher[numberOfEntries];
        try{
```

```
            //loop through the datafile
            for(int i=0; i<numberOfEntries; i++) {
                int entryToRead = i*RECORD_BYTES;

                publisherEntries[i] = readPublisherFileRecord(entryToRead);
            }
        } catch(Exception e) {
            //publisherFile does not exist or can not be created.
            String err = e.toString();
            System.out.println("");
            System.out.println("The publisher file does not exist or can not be created.");
            System.out.println("");
        }

        return publisherEntries;

    } catch (IOException io) {
        System.out.println("Error trying to open the publisher file.");
    }

    return null;
}

/**
 * readPublisherFileRecord - reads a publisher entry from the bookstore
 *
 * @param entryToRead   number of entry to read from the publisherFile
 */
public static Publisher readPublisherFileRecord(int entryToRead) throws IOException
{
    RandomAccessFile thePublisherFile;
    int numberOfEntries = 0;

    try{
        thePublisherFile = new RandomAccessFile(PUBLISHER_FILENAME, "rw");
        int publisherFileSize = (int) thePublisherFile.length();
        numberOfEntries = publisherFileSize / RECORD_BYTES;

        thePublisherFile.seek(entryToRead);
        int id = thePublisherFile.readInt();

        byte[] nameBytes = new byte[STRING_BYTES];
```

```
            thePublisherFile.readFully(nameBytes);
            String name = new String(nameBytes);

            byte[] addressBytes = new byte[STRING_BYTES];
            thePublisherFile.readFully(addressBytes);
            String address = new String(addressBytes);

            int telephone = thePublisherFile.readInt();

            byte[] websiteBytes = new byte[STRING_BYTES];
            thePublisherFile.readFully(websiteBytes);
            String website = new String(websiteBytes);

            return new Publisher(id, name, address, telephone, website);
        } catch (IOException io) {
            System.out.println("Error trying to open the publisher file.");
        }

        return null;
    }

    /**
     * writeToPublisherFile - writes a new publisher entry to the publisher file.
     *
     * @param  p    a new publisher entry
     */
    public static Boolean writeToPublisherFile(Publisher p)
    {
        RandomAccessFile thePublisherFile;
        int numberOfEntries = 0;

        try{
            thePublisherFile = new RandomAccessFile(PUBLISHER_FILENAME, "rw");
            int publisherFileSize = (int) thePublisherFile.length();
            numberOfEntries = publisherFileSize / RECORD_BYTES;

            try {
                int entryToWrite = numberOfEntries*RECORD_BYTES;
                writePublisherFileRecord(entryToWrite, p);
                numberOfEntries = numberOfEntries+1;
                return true;
```

```
            } catch (IOException io) {
                System.out.println("Error while writing to the publisher file.");

                return false;
            }
        } catch (IOException io) {
            System.out.println("Error trying to open the publisher file.");
        }

        return null;
    }

    /**
     * rewriteToPublisherFile - re-writes publisher entries to the bookstore
     *
     * @param entries          entries to write to the publisherFile
     * @return                 whether the write was successful
     */
    public static Boolean rewriteToPublisherFile(Publisher[] entries)
    {
        RandomAccessFile thePublisherFile;
        int numberOfEntries = 0;

        try{
            thePublisherFile = new RandomAccessFile(PUBLISHER_FILENAME, "rw");
            int publisherFileSize = (int) thePublisherFile.length();
            numberOfEntries = publisherFileSize / RECORD_BYTES;

            try {
                thePublisherFile.setLength(0);

                numberOfEntries = entries.length;

                for(int i=0; i<entries.length; i++) {
                    int entryToWrite = i*RECORD_BYTES;
                    writePublisherFileRecord(entryToWrite, entries[i]);
                }
                return true;

            } catch (IOException io) {
```

```
            System.out.println("Error while writing to the publisher file.");

            return false;
        }
    } catch (IOException io) {
        System.out.println("Error trying to open the publisher file.");
    }

    return null;
}

/**
 * writePublisherFileRecord - writes a publisher entry to the bookstore
 *
 * @param entryToWrite   number of entry to write to the publisherFile
 * @param entry          entry to write to the publisherFile
 */
public static void writePublisherFileRecord(int entryToWrite, Publisher entry) throws IOException
{
    RandomAccessFile thePublisherFile;
    int numberOfEntries = 0;

    try{
        thePublisherFile = new RandomAccessFile(PUBLISHER_FILENAME, "rw");
        int publisherFileSize = (int) thePublisherFile.length();
        numberOfEntries = publisherFileSize / RECORD_BYTES;

        int id = entry.getId();

        String name = entry.getName();
        StringBuffer sbtName = new StringBuffer(name);

        while(sbtName.length() < STRING_BYTES) {
            sbtName.append(" ");
        }

        name = sbtName.toString();

        String address = entry.getAddress();
        StringBuffer sbtAddress = new StringBuffer(address);
```

```
            while(sbtAddress.length() < STRING_BYTES) {
                sbtAddress.append(" ");
            }

            address = sbtAddress.toString();

            int telephone = entry.getTelephone();

            String website = entry.getWebsite();
            StringBuffer sbtWebsite = new StringBuffer(website);

            while(sbtWebsite.length() < STRING_BYTES) {
                sbtWebsite.append(" ");
            }

            website = sbtWebsite.toString();

            thePublisherFile.seek(entryToWrite);
            thePublisherFile.writeInt(id);
            thePublisherFile.write(name.getBytes());
            thePublisherFile.write(address.getBytes());
            thePublisherFile.writeInt(telephone);
            thePublisherFile.write(website.getBytes());

        } catch (IOException io) {
            System.out.println("Error trying to open the publisher file.");
        }
    }

    /**
     * readPublisherWithID - returns the publisher with a specific id.
     *
     * @param  id   the publisher id
     */
    public static Publisher readPublisherWithID(int id)
    {
        Publisher[] publishers = null;
        publishers = readPublisherFile();
        Publisher p = null;

        for(int i=0; i<publishers.length; i++) {
```

```
            if (publishers[i].getId() == id) {
                return p = publishers[i];
            }
        }

        return p;
    }

    /**
     * outputAllPublishersToTerminal - outputs all the publishers to the
     * terminal without sorting
     *
     */
    public static void outputAllPublishersToTerminal()
    {
        Publisher[] publishers = null;
        publishers = readPublisherFile();

        if(publishers != null) {

            for(int i=0; i < publishers.length; i++) {

                Publisher temp = publishers[i];

                int id = temp.getId();
                String name = temp.getName();
                String address = temp.getAddress();
                int telephone = temp.getTelephone();
                String website = temp.getWebsite();

                System.out.println("");
                System.out.println("================================");
                System.out.println("ID:" + id);
                System.out.println("Name:" + name);
                System.out.println("Address:" + address);
                System.out.println("Telephone:" + telephone);
                System.out.println("Website:" + website);
                System.out.println("================================");
                System.out.println("");
            }
```

```
    } else {
        System.out.println("");
        System.out.println("There are no books in the bookstore.");
        System.out.println("");
    }
}

/**
 * outputAllPublishersIDsNamesToTerminal - outputs all the publishers' ID and names
 * to the terminal without sorting
 *
 */
public static void outputAllPublishersIDsNamesToTerminal()
{
    Publisher[] publishers = null;
    publishers = readPublisherFile();

    if(publishers != null) {

        if(publishers.length == 0) {
            System.out.println("");
            System.out.println("The are no publishers.");
            System.out.println("");
        } else {

            System.out.println("");
            System.out.println("Available publishers:");
            System.out.println("================================");

            for(int i=0; i < publishers.length; i++) {

                Publisher temp = publishers[i];

                int id = temp.getId();
                String name = temp.getName();
                String address = temp.getAddress();
                int telephone = temp.getTelephone();
                String website = temp.getWebsite();

                System.out.println("ID:" + id + ", NAME:" + name);
            }
            System.out.println("================================");
            System.out.println("");
        }
```

```
        } else {
            System.out.println("");
            System.out.println("There are no books in the bookstore.");
            System.out.println("");
        }
    }
}
```

## Chapter References


1. International Baccalaureate Organization. (2012). IBDP Computer Science Guide.
2. Recursion (computer science). (27, May 2016). In *Wikipedia, The Free Encyclopedia*. Retrieved 13:39, June 3, 2016, https://en.wikipedia.org/wiki/Recursion_(computer_science)
3. Peter Smith. An iterative solution to Towers of Hanoi. Course Comp 151, California State University Northridge, September 2002. Retrieved 22:20, June 5, 2016, http://www.csun.edu/~psmith/151handouts/hanoi3out.pdf
4. Dimitriou K. Hatzitaskos M. (2015). Core Computer Science for the IB Diploma Program. Athens: Express Publishing. More information at: https://www.expresspublishing.co.uk/gr/en/content/core-computer-science-ib-diploma-program
5. Class ArrayList. In *Java Documentation*. Retrieved 19:00, July 8, 2016, https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
6. Class LinkedList. In *Java Documentation*. Retrieved 19:00, July 8, 2016, https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html
7. Robert L. Glass. Facts and Fallacies of Software Engineering, Addison Wesley (2003)
8. Indent style. (16, July 2016). In *Wikipedia, The Free Encyclopedia*. Retrieved 15:05, July 20, 2016, from https://en.wikipedia.org/wiki/Indent_style
9. Characters per line. (7, July 2016). In *Wikipedia, The Free Encyclopedia*. Retrieved 15:10, July 20, 2016, from https://en.wikipedia.org/wiki/Characters_per_line

# APPENDIX A – Stacks & Queues

## Appendix A — Stacks & Queues

### A.1 Stack implementation using the ArrayList class

```
//First file
import java.util.Scanner;

public class Main
{
    public static void main(String[] args) {
        boolean exit = false;
        Stack stack = new Stack();
        Scanner input = new Scanner(System.in);

        System.out.println("Stack manipulation program");

        while (exit == false) {
            System.out.println("Select one of the following options:");
            System.out.println("1. Push element to stack");
            System.out.println("2. Pop element from stack");
            System.out.println("3. Print the elements of the stack");
            System.out.println("4. Exit");
            System.out.print("User selection: ");

            int selection = input.nextInt();
            System.out.println();

            switch (selection) {
                case 1:
                System.out.print("Enter element: ");
                int element = input.nextInt();
                stack.push(element);
                break;
                case 2:
                stack.pop();
                break;
                case 3:
```

```
                stack.printElements();
                break;
                case 4:
                exit = true;
                break;
            }

            System.out.println();
        }
    }
}

//Second file
import java.util.ArrayList;

public class Stack {
    private ArrayList<Integer> Stack;

    public Stack() {
        Stack = new ArrayList();
    }

    public void push(int number) {
        Stack.add(number);
    }

    public void pop() {
        if (Stack.isEmpty()) {
            System.out.println("The stack is empty.");
        } else {
            int number = Stack.get(Stack.size() - 1);
            Stack.remove(Stack.size() - 1);
            System.out.println("Element removed: " + number);
        }
    }
```

```
    public void printElements() {
        if (Stack.isEmpty()) {
            System.out.println("The stack is empty.");
        } else {
            for (int i = Stack.size() - 1; i >= 0; i--) {
                System.out.println(Stack.get(i));
            }
        }
    }
}
```

## A.2 Queue implementation using the `ArrayList` class

```
//First file
import java.util.Scanner;

public class Main
{
    public static void main(String[] args) {
        boolean exit = false;
        Stack stack = new Stack();
        Scanner input = new Scanner(System.in);

        System.out.println("Stack manipulation program");

        while (exit == false) {
            System.out.println("Select one of the following options:");
            System.out.println("1. Push element to stack");
            System.out.println("2. Pop element from stack");
            System.out.println("3. Print the elements of the stack");
            System.out.println("4. Exit");
            System.out.print("User selection: ");
```

```
            int selection = input.nextInt();
            System.out.println();

            switch (selection) {
                case 1:
                System.out.print("Enter element: ");
                int element = input.nextInt();
                stack.push(element);
                break;
                case 2:
                stack.pop();
                break;
                case 3:
                stack.printElements();
                break;
                case 4:
                exit = true;
                break;
            }

            System.out.println();
        }
    }
}

//Second file
import java.util.ArrayList;

public class Stack {
    private ArrayList<Integer> Stack;

    public Stack() {
        Stack = new ArrayList();
    }

    public void push(int number) {
        Stack.add(number);
```

```
    }

    public void pop() {
        if (Stack.isEmpty()) {
            System.out.println("The stack is empty.");
        } else {
            int number = Stack.get(Stack.size() - 1);
            Stack.remove(Stack.size() - 1);
            System.out.println("Element removed: " + number);
        }
    }

    public void printElements() {
        if (Stack.isEmpty()) {
            System.out.println("The stack is empty.");
        } else {
            for (int i = Stack.size() - 1; i >= 0; i--) {
                System.out.println(Stack.get(i));
            }
        }
    }
}
```